IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Computational Interpretations of Logics

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/90204 since 2016-06-30T15:11:10Z

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

05 May 2023

IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

This is the author's final version of the contribution published as:

Silvia Ghilezan; Silvia Likavec. Computational Interpretations of Logics. ZBORNIK RADOVA. 12(20) pp: 159-215.

When citing, please refer to the published version.

Link to this full text:
http://hdl.handle.net/2318/90204

This full text was downloaded from iris - AperTO: https://iris.unito.it/

iris - AperTO

University of Turin's Institutional Research Information System and Open Access Institutional Repository

Silvia Ghilezan and Silvia Likavec

COMPUTATIONAL INTERPRETATIONS OF LOGICS

Abstract. The fundamental connection between logic and computation, known as the Curry–Howard correspondence or formulae-as-types and proofs-as-programs paradigm, relates logical and computational systems. We present an overview of computational interpretations of intuitionistic and classical logic:

- intuitionistic natural deduction - λ-calculus
- intuitionistic sequent calculus - λ^{Gtz}-calculus
- classical natural deduction - $\lambda\mu$-calculus
- classical sequent calculus - $\overline{\lambda}\mu\widetilde{\mu}$-calculus.

In this work we summarise the authors' contributions in this field. Fundamental properties of these calculi, such as confluence, normalisation properties, reduction strategies call-by-value and call-by-name, separability, reducibility method, λ-models are in focus. These fundamental properties and their counterparts in logics, via the Curry–Howard correspondence, are discussed.

Contents

Introduction

Gentzen's natural deduction is a well established formalism for expressing proofs. The simply typed λ-calculus of Church is a core formalism for writing programs. According to Curry–Howard correspondence, first formulated in 1969 by Howard [97], simply typed λ-calculus represents a computational interpretation of intuitionistic logic in natural deduction style and simplifying a proof corresponds to executing a program.

Griffin extended the Curry–Howard correspondence to classical logic in his seminal 1990 paper [90], by observing that classical tautologies suggest typings for certain control operators. The $\lambda\mu$-calculus of Parigot [117] expresses the content of classical natural deduction and has been the basis of a number of investigations into the relationship between classical logic and theories of control in programming languages [118, 40, 116, 19, 3]. At the same time proof-term calculi expressing a computational interpretation of classical logic serve as tools for extracting the constructive content of classical proofs [114, 6]. The recent interest in the Curry–Howard correspondence for sequent calculus [92, 12, 58, 56] made it clear that the computational content of sequent derivations and cut-elimination can be expressed through various extensions of the λ-calculus. There are several term calculi based on sequent calculus, in which reduction corresponds to cut elimination [93, 143, 34, 147, 103]. In contrast to natural deduction proof systems, sequent calculi exhibit inherent symmetries in proof structures which create technical difficulties in analyzing the reduction properties of these calculi.

In this work we summarise the authors' contributions in this field.

- **Part 1 – Background** gives a brief account on different formulations of intuitionistic and classical propositional logic as well as on λ-calculus and other proof-term calculi which express computational interpretations of logics.
  - Section 1 presents natural deduction and sequent calculus formulations of intuitionistic and classical propositional logic;
  - Section 2-5 present different term calculi that embody proofs in logics: the well-known λ-calculus of Church, $\lambda\mu$-calculus of Parigot [117], $\bar{\lambda}$-calculus of Herbelin [92], λ^{Gtz}-calculus of Espírito Santo [56] and $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [34];
  - Section 6 presents the fundamental relation between logic and computation, the Curry–Howard correspondence, which links formulae with types and proofs with terms and programs.
- **Part 2 – Contributions** has five sections and is the main part of this work, concentrating on the authors' contributions in each of the following fields:
  - Section 7 – *Intuitionistic natural deduction and λ-calculus*: summarises the results of Barendregt and Ghilezan [12], Ghilezan [71, 73, 72, 74, 76, 75, 77, 78, 79], Dezani, Ghilezan and Venneri [45], Ghilezan and Likavec [86, 87], Ghilezan and Kunčak [82, 83], Ghilezan, Kunčak

and Likavec [84], Likavec [105], Dezani and Ghilezan [41, 43, 42], and Dezani, Ghilezan and Likavec [44];
- Section 8 – *Intuitionistic sequent calculus and* λ^{Gtz}*-calculus*: summarises the results of Espírito Santo, Ghilezan and Ivetić [57], and Ghilezan and Ivetić [81];
- Section 9 – *Classical natural deduction and* $\lambda\mu$*-calculus*: summarises the results of Herbelin and Ghilezan [94];
- Section 10 – *Classical sequent calculus and* $\overline{\lambda}\mu\widetilde{\mu}$*-calculus*: summarises the results of Dougherty, Ghilezan and Lescanne [48, 49, 50, 51], Dougherty, Ghilezan, Lescanne and Likavec [52], and Likavec and Lescanne [107];
- Section 11 – *Application to functional and object-oriented programming languages*: summarises the results of Herbelin and Ghilezan [94], Likavec [106], and Bettini, Bono and Likavec [13, 14, 15, 16, 17, 18].

• **Part 3 – Related work** gives some pointers to the related work in the literature.

Part 1 – Background

1. Natural deduction and sequent calculus

In 1879 Gottlob Frege wrote his *Begriffsschrift* [66] paving a path for modern logic. Frege wanted to show that logic gave birth to mathematics. He invented axiomatic predicate logic, including quantified variables, adding iterations to the previous world of the logical constants *and, or, if... then..., not*, *some* and *all*. With Frege's "conceptual notation" inferences involving very complex mathematical statements could be represented. He formalised the rule of *modus ponens* using two kinds of judgements: premises and conclusions. Over time, Frege's pictorial notation (see [147] for an example of the original notation) evolved into the notation similar to the one we use today, namely $A \to B$ meaning "A implies B" and $\vdash A$ asserting "A is true". Here is the *modus ponens* rule using this notation

$$\frac{\vdash A \to B \qquad \vdash A}{\vdash B}$$

Axiomatic systems in the Hilbert tradition consist of axioms, modus ponens, and a few other inference rules. Another perspective to capture mathematical reasoning was to describe deduction through inference rules which explain the meaning of the logical connectives and quantifiers.

This giant step in formalizing logic was Gerhard Gentzen's *Unterschungen über das logische Schliessen* [69] written in 1935. In this work, Gentzen introduced the systems of *natural deduction* and *sequent calculus* for propositional and predicate logic, in both *intuitionistic* and *classical* variants. These two systems have the same set of derivable statements. In his work, Gentzen introduced assumptions, so his judgement had the following form:

$$A_1, \ldots, A_n \vdash B$$

meaning "Under the assumption that $A_1, \ldots, A_n$ are true we can conclude that B is true". Using this notation, the modus ponens rule is written as follows

$$\frac{\Gamma \vdash A \to B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

where Γ and Δ denote sequences of formulae.

1.1. **Natural deduction: intuitionistic logic NJ and classical logic NK.** We now present the two systems of Gentzen: *natural deduction* for intuitionistic logic, denoted by NJ, and classical logic, denoted by NK, as well as *sequent calculus* for intuitionistic logic, denoted by LJ, and classical logic, denoted by LK. For comprehensive account of the subject we refer the reader to [124].

The set of formulae of propositional logic is given by the following abstract syntax:

$$A, B \ = X \mid A \to B \mid A \wedge B \mid A \vee B \mid \neg A$$

where X denotes an atomic formula and capital Latin letters $A, B, C, \ldots$ denote formulae or single propositions. We will mostly deal with implicational formulae only and sometimes comment on other connectives. Hence, a formula can be one of the following: atomic formula X or implication $A \to B$. Capital Greek letters $\Gamma, \Delta, \ldots$ are used to denote sequences of formulae called antecedents and succedents. Γ, A stands for $\Gamma \cup \{A\}$.

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\text{axiom})$$

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\to \text{elim}) \qquad \frac{\Gamma,\ A \vdash B}{\Gamma \vdash A \to B}\ (\to \text{intro})$$

FIGURE 1. NJ: intuitionistic natural deduction

$$\frac{}{\Gamma, A \vdash A, \Delta}\ (\text{axiom})$$

$$\frac{\Gamma \vdash A \to B, \Delta \quad \Gamma \vdash A, \Delta}{\Gamma \vdash B, \Delta}\ (\to \text{elim}) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\ (\to \text{intro})$$

FIGURE 2. NK: classical natural deduction

The rules of Gentzen's natural deduction intuitionistic logic NJ and classical logic NK are given in Figures 1 and 2, respectively. Gentzen's system consists of structural and logical rules. The only structural rule is the axiom, whereas each of the connectives has introduction and elimination logical rules. Each introduction rule has the connective in the conclusion but not in the premises, whereas each elimination rule has the connective in the premises but not in the conclusion.

The following formulae are provable in classical logic, but not in intuitionistic:

- Pierce's law: $(A \to B) \to A \to A$
- Excluded middle: $A \vee \neg A$
- Double negation: $\neg\neg A \to A$.

The connection between logical connectives in classical logic and their dependencies is well known. As opposed to classical logic, connectives in intuitionistic logic are independent.

1.2. **Sequent calculus: intuitionistic logic LJ and classical logic LK.** Gentzen introduced the sequent calculus primarily as a tool to prove the consistency of predicate logic. In sequent calculus, a sequent has the form

$$A_1, \ldots, A_n \vdash B_1, \ldots, B_m \quad \text{or shorter} \quad \Gamma \vdash \Delta$$

which corresponds to the formula

$$A_1 \wedge \ldots \wedge A_n \to B_1 \vee \ldots \vee B_m.$$

For each connective, there are left and right logical rules, depending on whether the connective is introduced in antecedent or succedent. The rules of Gentzen's sequent calculus intuitionistic logic LJ and classical logic LK are given in Figures 3 and 4, respectively. Right rules in sequent calculus correspond to introduction rules in natural deduction, whereas left rules correspond to elimination rules. Both natural deduction and sequent calculus can be extended to incorporate other connectives, as well as quantifiers.

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\text{axiom})$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}\ (\to \text{left}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to \text{right})$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\ (\text{cut})$$

FIGURE 3. LJ: intuitionistic sequent calculus

$$\frac{}{\Gamma, A \vdash A, \Delta}\ (\text{axiom})$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\ (\to \text{left}) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\ (\to \text{right})$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\ (\text{cut})$$

FIGURE 4. LK: classical sequent calculus

The cut rule simplifies and shortens deductions, but at the same time makes it impossible to reconstruct the proofs, since we cannot know which formula was eliminated using the cut rule. Therefore, it is of uttermost importance to know that it is possible to leave out the cut rule and still obtain the system with the same set of derivable statements. This is exactly what Gentzen's *Cut elimination property (Hauptsatz)* proves.

Gentzen also formulated the *subformula property*: given a judgement $\Gamma \vdash A$, its proof can be simplified in such a way that only the propositions appearing in Γ and A and their subformulae appear in the proof of $\Gamma \vdash A$.

Theorem (Equivalence)**.**
A formula is derivable in NJ *if and only if it is derivable in* LJ*.*
A formula is derivable in NK *if and only if it is derivable in* LK*.*

2. λ-calculus

2.1. **Untyped λ-calculus.** The λ-calculus was originally formalised by Alonzo Church in 1932 [27] as a part of a general theory of functions and logic, in order to establish the limits of what was computable. Later on, it was shown that the full system was inconsistent. But the subsystem dealing with functions only proved to be a successful model for the computable functions and is called the *λ-calculus*.

The λ-calculus is a formal system that is meant to deal with functions and constructions of new functions. Expressions in this theory are called *λ-terms* and each such expression denotes a function. We denote the set of λ-terms by Λ.

Church developed a formalism for defining computable functions using three basic constructions: variables, λ-abstraction, and application, with one reduction rule. The formal syntax of λ-calculus is given by the following:

$$t ::= x \mid \lambda x.t \mid tt$$

where x is a variable, $\lambda x.t$ is a λ-abstraction (which represents a mapping $x \mapsto t$), and tt is the application (which represents application of a function to its argument). For comprehensive account of the subject we refer the reader to [10].

The set $Fv(t)$ of free variables of a λ-term t is defined inductively.

$$1.\ Fv(x) = \{x\} \quad 2.\ Fv(t_1t_2) = Fv(t_1) \cup Fv(t_2) \quad 3.\ Fv(\lambda x.t) = Fv(t)\backslash\{x\}.$$

The set Λ° of closed lambda terms is the set of lambda terms with no free variables

$$\Lambda^\circ = \{t \in \Lambda \mid Fv(t) = \emptyset\}.$$

The following reduction rule is called the *α-reduction*

$$\lambda x.t \to \lambda y.t[x := y],$$

where all the free occurrences of the variable x in t are replaced with a fresh variable y not occurring in t. The substitution $t_1[x := t_2]$ is not part of the syntax and it is defined so that all the free occurrences of the variable x in t_1 are replaced by t_2, taking into account that the free variables in t_2 remain free in the term obtained.

The main reduction rule of the λ-calculus is the *β-reduction*

$$(\lambda x.t_1)t_2 \to_\beta t_1[x := t_2].$$

A λ-term of the form $(\lambda x.t_1)t_2$ is called a *redex*. The transitive reflexive contextual closure of $\rightarrow_\beta$ is denoted by $\twoheadrightarrow_\beta$. The β-equality $=_{(\beta)}$ (β-conversion) is the symmetric transitive closure of $\twoheadrightarrow_\beta$.

The *η-reduction* is given by

$$\lambda x.tx \rightarrow_\eta t, \quad x \notin Fv(t),$$

where $\lambda x.tx$ is called an *η-redex*, provided that $x \notin Fv(t)$. The transitive reflexive closure of $\rightarrow_\eta$ is denoted by $\twoheadrightarrow_\eta$. The η-equality $=_{(\eta)}$ is the symmetric transitive closure of $\twoheadrightarrow_\eta$. The reductions β and η together generate a reduction denoted by $\twoheadrightarrow$.

This simple syntax equipped with simple reduction rules gives rise to a powerful formal system which is Turing complete. The functions representable in λ-calculus coincide with Turing computable functions and recursive functions.

We give now some of the basic notions that we will use later.

- If $t \equiv \lambda x_1 \dots x_n.(\lambda x.t_0)t_1 \dots t_m$, $n \geqslant 0$, $m \geqslant 1$, then $(\lambda x.t_0)t_1$ is called the *head-redex* of t (Barendregt [10, p. 173]). We write $t \rightarrow_h t'$ if t' is obtained from t by reducing the head redex of t (head reduction). We write $t \rightarrow_i t'$ if t' is obtained from t by reducing a redex other than the head redex (internal reduction). We also use the transitive closures of these relations, notation $\twoheadrightarrow_h$ and $\twoheadrightarrow_i$, respectively.
- A term is a *normal form* if it does not contain any redex. A term is *normalising* (has a normal form) if it reduces to a normal form. The set of all λ-terms that have a normal form will be denoted by $\mathcal{N}$. All normal forms are of the form:

  $$\lambda y_1 \dots y_n.zt_1 \dots t_k,$$

  where t_i, $1 \leqslant i \leqslant k$, $0 \leqslant k$, are again normal forms, and z can be one of y_j, $1 \leqslant j \leqslant n$, $0 \leqslant n$.
- A term is *strongly normalising* if all its reduction paths end in a normal form (are finite). $\mathcal{SN}$ will denote the set of strongly normalising terms, i.e.,

  $$\mathcal{SN} = \{t \in \Lambda \mid \neg(\exists t_1, t_2, \dots \in \Lambda)\, t \rightarrow_\beta t_1 \rightarrow_\beta t_2 \rightarrow_\beta \dots\}.$$

- A *head normal form* is a term of the form

  $$\lambda x_1 \dots x_n.yt_1 \dots t_l,$$

  where y can be one of x_i, $1 \leqslant i \leqslant n, 0 \leqslant n$ and $t_j \in \Lambda$, $1 \leqslant j \leqslant l, 0 \leqslant l$.
- A term t is *solvable* (*has a head normal form*) if there exists $t' \in \Lambda$ such that $t \rightarrow t'$ and t' is a head normal form. The set of all solvable λ-terms is denoted by $\mathcal{HN}$. A term is *unsolvable* if it is not solvable.
- A term is a *weak head normal form* if it starts with an abstraction, or with a variable. A term is *weakly head normalising* (has a weak head normal form) if it reduces to a weak head normal form. The set of all λ-terms that have a weak head normal form will be denoted by $\mathcal{WN}$.

  $$\mathcal{WN} = \{t \in \Lambda \mid (\exists t', t_1, \dots, t_n \in \Lambda)\, t \twoheadrightarrow_\beta \lambda x.t' \text{ or } t \twoheadrightarrow_\beta xt_1 \dots t_n\}.$$

- *Church-Rosser theorem (Confluence)*: If $t_1 \twoheadleftarrow t \twoheadrightarrow t_2$, then there exists a λ-term $t_3 \in \Lambda$ such that $t_1 \twoheadrightarrow t_3 \twoheadleftarrow t_2$.

2.2. **Typed λ-calculus.** In 1940 Church formulated *typed λ-calculus* [28] as a way to avoid the paradoxes existing in other logics. Types are syntactical objects assigned to λ-terms in order to specify the properties of these λ-terms.

The basic type assignment system is the *simply typed λ-calculus* $\lambda \rightarrow$, or Curry's type inference system. The types in this system are formed using only the arrow operator $\rightarrow$. The application of λ-terms yields the arrow elimination on types, while the abstraction yields the arrow introduction.

The set Type of types is defined as follows.

$$A, B \ ::= \ X \mid A \rightarrow B$$

where X ranges over a denumerable set $TVar$ of type atoms.

The following notions will be used in our work:

- A *type assignment* is an expression of the form $t : A$, where $t \in \Lambda$ and $A \in \mathsf{Type}$.
- A *context* (*basis*) Γ is a set $\{x_1 : A_1, \ldots, x_n : A_n\}$ of type assignments with different term variables, $Dom\,\Gamma = \{x_1, \ldots, x_n\}$ and $\Gamma \setminus \mathbf{x} = \{A_1, \ldots, A_n\}$. We use capital Greek letters $\Gamma, \Delta, \Gamma_1, \ldots$ to denote contexts.
- A context extension $\Gamma, x : A$ denotes the set $\Gamma \cup \{x : A\}$, where $x \notin Dom\Gamma$.

The type assignment $t : B$ is derivable from the context Γ in the type system $\lambda \rightarrow$, notation $\Gamma \vdash t : B$, if $\Gamma \vdash t : B$ can be generated by the axiom and rules given in Figure 5.

$$\frac{}{\Gamma, x : A \vdash x : A}\,(ax)$$

$$\frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}\,(\rightarrow E) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B}\,(\rightarrow I)$$

FIGURE 5. $\lambda \rightarrow$: simply typed λ-calculus

We list some of the most important properties of $\lambda \rightarrow$. The property of preservation of types under reduction is referred to as *Subject reduction*.

Theorem (Subject reduction)**.** *If $\Gamma \vdash t : A$ and $t \twoheadrightarrow u$, then $\Gamma \vdash u : A$.*

An important property, which might be a reason for introducing types in λ-calculus, is the *strong normalisation* of all typeable terms.

Theorem (Strong normalisation)**.** *If a term is typeable in $\lambda \rightarrow$, then it is strongly normalising.*

The correspondence between formulae of intuitionistic logic NJ and types of $\lambda \rightarrow$, together with the correspondence of proofs in NJ and terms of $\lambda \rightarrow$, was given by Howard [97] based on earlier work of Curry. It is nowadays referred to as the *Curry–Howard correspondence* between logic and computation.

Theorem (Curry–Howard correspondence)**.** $\Gamma \vdash t : A$ *if and only if* $\Gamma \smallsetminus \mathbf{x} \vdash A$ *is derivable in* NJ.

There are many known extensions of $\lambda \rightarrow$. Extensions with polymorphic types and dependent types fit perfectly in the so called Barendregt's cube. For comprehensive account of the subject we refer the reader to [9].

2.3. **Intersection types for λ-calculus.** The extension of $\lambda \rightarrow$ which characterises exactly the strongly normalising terms is with intersection types, which are also suitable for analysing λ models and various normalisation properties of λ-terms. The intersection type assignment systems are originated by Coppo and Dezani [29, 30], Barendregt et al. [11], Copo et al. [31], Pottinger [123], and Sallé [130]. In this system, the new type-forming operator is introduced, the intersection $\cap$, whose properties are in accordance with its interpretation as intersection of types. Consequently, it is possible to assign two types A and B to a certain λ-term at the same time. Another outstanding feature of this system is the universal type Ω which can be assigned to all λ-terms. Therefore the question of typability is trivial in these systems.

We focus on the intersection type assignment system $\lambda\cap^{\Omega}$ with the type Ω. The set Type of types in $\lambda\cap^{\Omega}$ is defined as follows

$$A, B \ ::= \ X \mid \Omega \mid A \rightarrow B \mid A \cap B$$

where X ranges over a denumerable set $TVar$ of type atoms. A type assignment, a context, and a context extension are defined as usual.

The *preorder* on Type is defined in the following way:

(i) The relation $\leqslant$ is defined on Type by the following axioms and rules:

1. $A \leqslant A$
2. $A \leqslant B,\ B \leqslant C \ \Rightarrow A \leqslant C$
3. $A \cap B \leqslant A,\ A \cap B \leqslant B$
4. $(A \rightarrow B) \cap (A \rightarrow C) \leqslant A \rightarrow B \cap C$
5. $A \leqslant B,\ A \leqslant C \Rightarrow A \leqslant B \cap C$
6. $A \leqslant A',\ B \leqslant B' \Rightarrow A \cap B \leqslant A' \cap B'$
7. $A \leqslant A',\ B \leqslant B' \Rightarrow A' \rightarrow B \leqslant A \rightarrow B'$
8. $A \leqslant \Omega$
9. $A \rightarrow \Omega \leqslant \Omega \rightarrow \Omega$.

(ii) The induced equivalence relation is defined by:

$$A \sim B \Leftrightarrow A \leqslant B \ \& \ B \leqslant A.$$

The usual axiom of the preorder on intersection types is $\Omega \leqslant \Omega \rightarrow \Omega$ (Barendregt et al. [11]). Having this axiom one can distinguish head normalising terms from unsolvable terms by their typeability, but cannot distinguish weakly head normalising terms from unsolvable terms. Instead we adopt the axiom $A \rightarrow \Omega \leqslant \Omega \rightarrow \Omega$, which allows us to distinguish weakly head normalising from unsolvable terms (Dezani et al. [47]).

The type assignment $t : B$ is derivable from the context Γ in the type system $\lambda\cap^{\Omega}$, notation $\Gamma \vdash t : B$, if $\Gamma \vdash t : B$ can be generated by the axioms and rules given in Figure 6.

$$\frac{}{\Gamma, x : A \vdash x : A}\,(ax) \qquad \frac{}{\Gamma \vdash t : \Omega}\,(\Omega)$$

$$\frac{\Gamma \vdash t_1 : A \to B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}\,(\to E) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\to I)$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B}\,(\cap I) \qquad \frac{\Gamma \vdash t : A, \quad A \leqslant B}{\Gamma \vdash t : B}\,(\leqslant)$$

FIGURE 6. $\lambda\cap^{\Omega}$: intersection type assignment system

The following rule is derivable from the rules given in Figure 6:

$$\frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A\ (B)}\,(\cap E).$$

Some of the type assignment systems that can be obtained by combining the rules above and can be regarded as restrictions of $\lambda\cap^{\Omega}$ are given by the following axioms and rules:

- $\lambda\cap$: (ax), $(\to E)$, $(\to I)$, $(\cap E)$, $(\cap I)$, and $(\leqslant)$.
- $\mathcal{D}$: (ax), $(\to E)$, $(\to I)$, $(\cap E)$, and $(\cap I)$.
- $\mathcal{D}\Omega$: (ax), $(\to E)$, $(\to I)$, $(\cap E)$, $(\cap I)$, and (Ω).

All the eight typed calculi of Barendregt's cube satisfy the strong normalisation property, meaning that typeability in the system implies strong normalisation. A unique property of the two intersection type systems without Ω, namely $\lambda\cap$ and $\mathcal{D}$, is the inverse of strong normalisation property. In these systems all strongly normalising terms are typeable. Thus terms typeable in these systems coincide with strongly normalising terms. This outstanding property of intersection type systems has merited a lot of attention and has been proven by different authors and different means in [123, 31, 145, 73, 1], the list is not complete.

Theorem (Strong normalisation)**.** *The calculi $\lambda\cap$ and $\mathcal{D}$ satisfy the following*

$$t \text{ is typable } \Leftrightarrow t \text{ is strongly normalising.}$$

There are many known extensions of the λ-calculus with intersection types: Lengrand's et al. calculus with explicit substitutions [104], Matthes's calculus with generalised applications [109], Dougherty's et al. calculus for classical logic [51], Carlier and Wells's and Kfoury and Wells's calculi with expansion variables for type inference [26, 99], Dunfield and Pfenning's calculus with intersection, union, indexed, and universal and existential dependent types [54], to name just a few.

3. λ^{Gtz}-calculus

3.1. **Syntax and reduction rules.** There were several attempts, over the years, to design a term calculus which would embody the Curry–Howard correspondence

for intuitionistic sequent calculus. The first calculus accomplishing this task is Herbelin's $\bar{\lambda}$-calculus given in [92]. Recent interest in the Curry–Howard correspondence for sequent calculus [92, 12, 58, 56] made it clear that the computational content of sequent derivations and cut-elimination can be expressed through an extension of the λ-calculus. The λ^{Gtz}-calculus was proposed by Espírito Santo [56] as a modification of Herbelin's $\bar{\lambda}$-calculus. Its simply typed version corresponds to the sequent calculus for implicational fragment of intuitionistic logic.

The abstract syntax of λ^{Gtz} is given by:

$$\begin{array}{lrcl} \text{(Terms)} & t, u, v & ::= & x \mid \lambda x.t \mid tk \\ \text{(Contexts)} & k & ::= & \widehat{x}.t \mid u :: k. \end{array}$$

Terms are either variables, abstractions or *cuts* tk. A context is either a *selection* or a *context cons(tructor)*. According to the form of k, a cut may be an explicit substitution $t(\widehat{x}.v)$ or a multiary generalised application $t(u_1 :: \cdots u_m :: \widehat{x}..v)$, $m \geqslant 1$. In the last case, if $m = 1$, we get a generalised application $t(u :: \widehat{x}.v)$; if $v = x$, we get a multiary application $t[u_1, \cdots, u_m]$ ($\widehat{x}.x$ can be seen as the empty list of arguments).

In $\lambda x.t$ and $\widehat{x}.t$, t is the scope of the binders λx and $\widehat{x}$, respectively. The scope of binders extends to the right as much as possible.

Reduction rules of λ^{Gtz} are the following:

$$\begin{array}{lrcl} (\beta) & (\lambda x.t)(u :: k) & \to & u(\widehat{x}.tk) \\ (\pi) & (tk)k' & \to & t(k@k') \\ (\sigma) & t\widehat{x}.v & \to & v[x := t] \\ (\mu) & \widehat{x}.xk & \to & k, \text{ if } x \notin k \end{array}$$

where $v[x := t]$ denotes meta-substitution defined as usual, and $k@k'$ is defined by

$$(u :: k)@k' = u :: (k@k') \qquad (\widehat{x}.t)@k' = \widehat{x}.tk'.$$

The rules β, π, and σ reduce cuts to the trivial form $y(u_1 :: \cdots u_m :: \widehat{x}.v)$, for some $m \geqslant 1$, which represents a sequence of left introductions. Rule β generates a substitution, and rule σ executes a substitution on the meta-level. Rule π generalises the permutative conversion of the λ-calculus with generalised applications. Rule μ has a structural character, and it either performs a trivial substitution in the reduction $t(\widehat{x}.xk) \to tk$, or it minimises the use of the generality feature in the reduction $t(u_1 \cdots u_m :: \widehat{x}.xk) \to t(u_1 \cdots u_m :: k)$.

3.2. **Simply typed λ^{Gtz}-calculus.** The set Type of types, ranged over by $A, B, C, \ldots, A_1, \ldots$, is defined inductively:

$$A, B \ ::= \ X \mid A \to B$$

where X ranges over a denumerable set $TVar$ of type atoms.

There are two kinds of type assignment:

- $\Gamma \vdash t : A$ for typing terms;
- $\Gamma; B \vdash k : A$ for typing contexts.

The special place between the symbols ; and $\vdash$ is called the *stoup* and was proposed by Girard [89]. Stoup contains a selected formula, the one with which we continue computation.

The type assignment system $\lambda^{\mathsf{Gtz}} \to$ is given in Figure 7.

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Ax)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ (\to_R) \qquad \frac{\Gamma \vdash t : A \quad \Gamma; B \vdash k : C}{\Gamma; A \to B \vdash t :: k : C}\ (\to_L)$$

$$\frac{\Gamma \vdash t : A \quad \Gamma; A \vdash k : B}{\Gamma \vdash tk : B}\ (Cut) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma; A \vdash \widehat{x}.t : B}\ (Sel)$$

FIGURE 7. $\lambda^{\mathsf{Gtz}} \to$: simply typed λ^{Gtz}-calculus

4. $\lambda\mu$-calculus

4.1. **Syntax and reduction rules.** The original version of the typed $\lambda\mu$-calculus was formulated by Parigot [117] as the extension of λ-calculus with certain sequential operators and was meant to provide a proof term assignment for classical logic in natural deduction style. As said in [19], "$\lambda\mu$-calculus is a typed λ-calculus which is able to save and restore the runtime environment."

The $\lambda\mu$-calculus was introduced as a call-by-name language, but it received a call-by-value interpretation by Ong and Stewart in [116].

The syntax of the $\lambda\mu$-calculus is given by the following:

$$\begin{array}{lrcl} \text{unnamed terms:} & t & ::= & x \mid \lambda x.t \mid tu \mid \mu\beta.c \\ \text{named terms:} & c & ::= & [\alpha]t. \end{array}$$

We distinguish two kinds of variables: λ-variables $(x, y, \ldots x_1, \ldots)$ and μ-variables $(\alpha, \beta, \ldots\ \alpha_1, \ldots)$. We also distinguish two kinds of terms: *named* and *unnamed* terms. Named terms enable us to name arbitrary subterms by μ-variables and refer to them later.

The reduction rules of the $\lambda\mu$-calculus are:

$$\begin{array}{rcl} (\lambda x.u)t & \to & u[x := t] \\ (\mu\beta.c)t & \to & \mu\beta.c[[\beta]w := [\beta](wt)] \\ [\alpha](\mu\beta.c) & \to & c[\beta := \alpha]. \end{array}$$

In the second rule, every subterm of c of the form $[\beta]w$ is replaced by a term $[\beta](wt)$.

4.2. **Simply typed $\lambda\mu$-calculus.** The original version of the $\lambda\mu$-calculus is typed.

A type assignment $t : A$ is derivable from the contexts Γ and Δ in the system $\lambda\mu$, notation

$$\Gamma \vdash t : A, \Delta$$

if $\Gamma \vdash t : A, \Delta$ can be generated by the following axiom and rules given in Figure 8.

$$\frac{}{\Gamma, y : A \vdash y : A, \Delta}\ (\text{axiom})$$

$$\frac{\Gamma \vdash u : A \to B, \Delta \quad \Gamma \vdash t : A, \Delta}{\Gamma \vdash ut : B, \Delta}\ (\to \text{elim}) \qquad \frac{\Gamma, y : A \vdash u : B, \Delta}{\Gamma \vdash \lambda y.u : A \to B, \Delta}\ (\to \text{intro})$$

$$\frac{\Gamma \vdash u : A, \Delta, \beta : A, \alpha : B}{\Gamma \vdash \mu\alpha.[\beta]u : B, \Delta, \beta : A}\ (\mu)$$

FIGURE 8. Simply typed $\lambda\mu$-calculus

The typed calculus is both, strongly normalising and confluent and the types are preserved by the reduction.

5. $\overline{\lambda}\mu\widetilde{\mu}$-calculus

5.1. **Syntax and reduction rules.** The $\overline{\lambda}\mu\widetilde{\mu}$-calculus was introduced by Curien and Herbelin in [34].

The untyped version of the calculus can be seen as the foundation of a functional programming language with explicit notion of control and was further studied by Dougherty, Ghilezan, and Lescanne in [85, 48, 49, 51].

The syntax of $\overline{\lambda}\mu\widetilde{\mu}$ is given by the following, where v ranges over the set Term of terms, e ranges over the set Coterm of coterms and c ranges over the set $\mathsf{Command}$ of commands:

$$t ::= x \mid \lambda x\,.\,t \mid \mu\alpha\,.\,c \qquad e ::= \alpha \mid t \bullet e \mid \widetilde{\mu}x\,.\,c \qquad c ::= \langle t \parallel e \rangle.$$

There are two kinds of variables in the calculus: the set Var_v of variables (denoted by Latin letters $x, y, \ldots$) and the set Var_e of covariables (denoted by Greek letters $\alpha, \beta, \ldots$). The variables can be bound by λ-abstraction or by $\widetilde{\mu}$-abstraction, whereas the covariables can be bound by μ-abstraction. The sets of free variables and covariables, Fv_t and Fv_e, are defined as usual, respecting Barendregt's convention [10] that no variable can be both, bound and free, in the expression.

Terms yield values, while coterms consume values. A command is a cut of a term against a coterm. Commands are the place where terms and coterms interact. The components can be nested and more processes can be active at the same time.

There are only three rules that characterise the reduction in $\overline{\lambda}\mu\widetilde{\mu}$:

$$\begin{array}{lrcl} (\to') & \langle \lambda x\,.\,t_1 \parallel t_2 \bullet e \rangle & \to & \langle t_2 \parallel \widetilde{\mu}x.\langle t_1 \parallel e \rangle \rangle \\ (\mu) & \langle \mu\alpha\,.\,c \parallel e \rangle & \to & c[\alpha := e] \\ (\widetilde{\mu}) & \langle t \parallel \widetilde{\mu}x\,.\,c \rangle & \to & c[x := t]. \end{array}$$

The above substitutions are defined as to avoid variable capture [10].

As a rewriting calculus $\overline{\lambda}\mu\widetilde{\mu}$ has a critical pair $\langle \mu\alpha \,.\, c_1 \parallel \widetilde{\mu}x \,.\, c_2\rangle$ where both, (μ) and $(\widetilde{\mu})$ rule can be applied non-deterministically, producing two different results. For example,

$$\langle \mu\alpha.\langle y \parallel \beta\rangle \parallel \widetilde{\mu}x.\langle z \parallel \gamma\rangle\rangle \to_{\mu} \langle y \parallel \beta\rangle \quad \text{and} \quad \langle \mu\alpha.\langle y \parallel \beta\rangle \parallel \widetilde{\mu}x.\langle z \parallel \gamma\rangle\rangle \to_{\widetilde{\mu}} \langle z \parallel \gamma\rangle,$$

where α and β denote syntactically different covariables.

Hence, the calculus is not confluent. But if the priority is given either to (μ) or to $(\widetilde{\mu})$ rule, we obtain two confluent subcalculi $\overline{\lambda}\mu\widetilde{\mu}_T$ and $\overline{\lambda}\mu\widetilde{\mu}_Q$. There are two possible reduction strategies in the calculus that depend on the orientation of the critical pair. If the priority is given to (μ) redexes, call-by-value reduction is obtained ($\overline{\lambda}\mu\widetilde{\mu}_Q$-calculus), whereas giving the priority to $(\widetilde{\mu})$ redexes, simulates call-by-name reduction ($\overline{\lambda}\mu\widetilde{\mu}_T$-calculus).

This is more than simply a reflection of the well-known fact that the equational theories of call-by-name and call-by-value differ. It is a reflection of the great expressive power of the language: a single expression containing several commands can encompass several complete computational processes, and the μ and $\widetilde{\mu}$ reductions allow free transfer of control between them.

We first give the syntactic constructs of $\overline{\lambda}\mu\widetilde{\mu}_T$ and $\overline{\lambda}\mu\widetilde{\mu}_Q$, respectively:

$\overline{\lambda}\mu\widetilde{\mu}_T$		$\overline{\lambda}\mu\widetilde{\mu}_Q$	
$c ::=$	$\langle t \parallel e\rangle$	$c ::=$	$\langle t \parallel e\rangle$
$t ::=$	$x \mid \lambda x \,.\, t \mid \mu\alpha \,.\, c$	$V ::=$	$x \mid \lambda x \,.\, t$
$E ::=$	$\alpha \mid t \bullet E$	$t ::=$	$\mu\alpha \,.\, c \mid V$
$e ::=$	$\widetilde{\mu}x \,.\, c \mid E$	$e ::=$	$\alpha \mid \widetilde{\mu}x \,.\, c \mid V \bullet e.$

In $\overline{\lambda}\mu\widetilde{\mu}_T$ the new syntactic subcategory E of coterms, called *applicative contexts*, is introduced, in order to model call-by-name reduction. In $\overline{\lambda}\mu\widetilde{\mu}_Q$, notice the presence of the new syntactic construct V that models the *values*.

The reduction rules for $\overline{\lambda}\mu\widetilde{\mu}_T$ and $\overline{\lambda}\mu\widetilde{\mu}_Q$ are the following:

$\overline{\lambda}\mu\widetilde{\mu}_T$

$(\to)$	$\langle \lambda x \,.\, t_1 \parallel t_2 \bullet E\rangle$	$\to$	$\langle t_1[x \leftarrow t_2] \parallel E\rangle$
(μ)	$\langle \mu\alpha \,.\, c \parallel E\rangle$	$\to$	$c[\alpha := E]$
$(\widetilde{\mu})$	$\langle t \parallel \widetilde{\mu}x \,.\, c\rangle$	$\to$	$c[x := t]$

$\overline{\lambda}\mu\widetilde{\mu}_Q$

$(\to')$	$\langle \lambda x \,.\, t_1 \parallel V_2 \bullet e\rangle$	$\to$	$\langle V_2 \parallel \widetilde{\mu}x.\langle t_1 \parallel e\rangle\rangle$
(μ)	$\langle \mu\alpha \,.\, c \parallel e\rangle$	$\to$	$c[\alpha := e]$
$(\widetilde{\mu})$	$\langle V \parallel \widetilde{\mu}x \,.\, c\rangle$	$\to$	$c[x := V].$

Notice that in [34] only the rule $(\to')$ is considered for both subcalculi. In [85, 48, 49, 51] only the rule $(\to)$ is used. In [107, 106] $(\to)$ reduction is used rather than $(\to')$ reduction in the case of $\overline{\lambda}\mu\widetilde{\mu}_T$, since the application of the $(\to')$ rule will always be immediately followed by the application of the $(\widetilde{\mu})$ rule and that is exactly the rule $(\to)$. This choice makes explicit the priorities of the rules in each subcalculus.

In their original work on the $\overline{\lambda}\mu\widetilde{\mu}$-calculus [34], Curien and Herbelin defined a call-by-name and a call-by-value cps-translations of the complete *typed* $\overline{\lambda}\mu\widetilde{\mu}$-calculus into simply typed λ-calculus. The important point to notice is that they also interpret the types of the form $A-B$, which are dual to the arrow types $A \to B$. The translations validate call-by-name and call-by-value discipline, respectively.

In addition, as described in [34], the sequent calculus basis for $\overline{\lambda}\mu\widetilde{\mu}$ supports the interpretation of the reduction rules of the system as operations of an abstract machine. In particular, the right- and left-hand sides of a sequent directly represent the *code* and *environment* components of the machine. This perspective is elaborated more fully in [32]. See [33] for a discussion of the importance of symmetries in computation.

5.2. **Simply typed $\overline{\lambda}\mu\widetilde{\mu}$-calculus.** The set Type of types for the $\overline{\lambda}\mu\widetilde{\mu}$-calculus is obtained by closing a set of base types X under implication

$$A, B \ ::= \ X \mid A \to B.$$

Type bases have two components, the *antecedent* a set of bindings of the form $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, and the *succedent* of the form $\Delta = \alpha_1 : B_1, \ldots, \alpha_k : B_k$, where x_i, α_j are distinct for all $i = 1, \ldots, n$ and $j = 1, \ldots, k$. The judgements of the type system are given by the following:

$$\Gamma \ \vdash \ r : A \ \mid \ \Delta \qquad \Gamma \ \mid \ e : A \ \vdash \ \Delta \qquad c \ : \ (\Gamma \ \vdash \ \Delta)$$

where Γ is the antecedent and Δ is the succedent. The first judgement is a typing for a term, the second one is a typing for a coterm and the third one is a typing for a command. The box denotes a distinguished output or input, i.e., a place where the computation will continue or where it happened before. The type assignment system for the $\overline{\lambda}\mu\widetilde{\mu}$-calculus, introduced by Curien and Herbelin in [34], is given in Figure 9.

$$\frac{}{\Gamma, x : A \ \vdash \ x : A \ \mid \ \Delta}\,(axR) \qquad \frac{}{\Gamma \ \mid \ \alpha : A \ \vdash \ \alpha : A, \Delta}\,(axL)$$

$$\frac{\Gamma \ \vdash \ r : A \ \mid \ \Delta \qquad \Gamma \ \mid \ e : B \ \vdash \ \Delta}{\Gamma \ \mid \ r \bullet e : A \to B \ \vdash \ \Delta}\,(\to L) \qquad \frac{\Gamma, x : A \ \vdash \ r : B \ \mid \ \Delta}{\Gamma \ \vdash \ \lambda x.r : A \to B \ \mid \ \Delta}\,(\to R)$$

$$\frac{c \ : \ (\Gamma \ \vdash \ \alpha : A, \Delta)}{\Gamma \ \vdash \ \mu\alpha.c : A \ \mid \ \Delta}\,(\mu) \qquad \frac{c \ : \ (\Gamma, x : A \ \vdash \ \Delta)}{\Gamma \ \mid \ \widetilde{\mu}x.c : A \ \vdash \ \Delta}\,(\widetilde{\mu})$$

$$\frac{\Gamma \ \vdash \ r : A \ \mid \ \Delta \qquad \Gamma \ \mid \ e : A \ \vdash \ \Delta}{\langle r \parallel e \rangle \ : \ (\Gamma \ \vdash \ \Delta)}\,(cut)$$

FIGURE 9. Simply typed $\overline{\lambda}\mu\widetilde{\mu}$-calculus

6. **Curry–Howard correspondence**

The fundamental connection between logic and computation is given by Curry–Howard correspondence or formulae-as-types, proofs-as-terms, proofs-as-programs interpretation. It relates many computational and logical systems and can be applied to intuitionistic and classical logic, to sequent calculus and natural deduction.

Under the traditional Curry–Howard correspondence formulae provable in intuitionistic natural deduction coincide with types inhabited in simply typed λ-calculus. This was observed already by Curry, first formulated by Howard in 1969 [97], used extensively by de Bruijin in the Automath project and by Lambek in category theory. This correspondence extends to all eight calculi of Barendregt's cube and corresponding logical systems. We refer the reader to [135] for an extensive account of this topic.

Only in 1990 Griffin [90] showed that this correspondence can be extended to classical logic, pointing out that classical tautologies suggest typings for certain control operators: the Pierce's Law corresponds to the type of `call-cc` operator in Scheme (introduced by Sussman and Steele [139]) and the Law of Double Negation corresponds to the type of $\mathcal{C}$ operator (introduced by Felleisen et al. [61, 62]).

Extensive research in both natural deduction and sequent calculus formulations of classical logic followed. One of the cornerstones is the $\lambda\mu$-calculus of Parigot [117] which corresponds to classical natural deduction. It was followed by term calculi designed to incorporate classical sequent calculus: the Symmetric Lambda Calculus of Barabanera and Berardi [6], the $\overline{\lambda}\mu\widetilde{\mu}$-calculus of Curien and Herbelin [34], and the Dual calculus of Wadler [147, 148].

Part 2 – Contributions

In this part we give an overview of the work done by the authors in the field of computational interpretations of logics. In Section 7 we focus on intuitionistic natural deduction and the λ-calculus. In Section 8 we deal with intuitionistic sequent calculus and the λ^{Gtz}-calculus of [56]. In Sections 9 and 10 we concentrate on classical logic: the $\lambda\mu$-calculus [117], proof term assignment for classical natural deduction; and three proof term calculi for classical sequent calculus: the $\overline{\lambda}\mu\widetilde{\mu}$-calculus [34], the dual calculus [147, 148] and the Symmetric Lambda Calculus [6]. Finally, in Section 11 we turn to application to programming language theory.

7. Intuitionistic natural deduction and λ-calculus

7.1. **Terms for natural deduction and sequent calculus intuitionistic logic.** The correspondence between sequent calculus derivations and natural deduction derivations is not a one-to-one map: several cut-free derivations correspond to one normal derivation. In Barendregt and Ghilezan [12] this is explained by two extensionally equivalent type assignment systems for untyped λ-terms, namely λN and λL, one corresponding to intuitionistic natural deduction NJ and the other to intuitionistic sequent calculus LJ. These two systems constitute different grammars for generating the same (type assignment relation for untyped) λ-terms. Moreover, the second type assignment system has a 'cut-free' fragment (λL^{cf}) which generates exactly the typeable λ-terms in normal form. The cut elimination theorem becomes a simple consequence of the fact that typed λ-terms posses a normal form.

There are three type systems that assign types to untyped λ-terms:

- λN is the simply typed λ-calculus, $\lambda\!\rightarrow$, given in Figure 5;
- λL given in Figure 10;
- λL^{cf}, the cut-free fragment of λL (rules of Figure 10 without the (cut) rule).

The last two systems have been described by Gallier [68], Barbanera et al. [8], and Mints [110]. The three systems λN, λL, and λL^{cf} correspond exactly to the intuitionistic natural deduction NJ (Figure 1), the intuitionistic sequent calculus LJ (Figure 3), and the cut-free fragment of LJ. We denote NJ, LJ, and cut-free LJ by N, L and L^{cf} respectively.

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A}\ (\text{axiom})$$

$$\frac{\Gamma\vdash s:A \quad \Gamma, x:B\vdash t:C}{\Gamma, y:A\to B\vdash t[x:=ys]:C}\ (\to \text{left}) \qquad \frac{\Gamma, x:A\vdash t:B}{\Gamma\vdash \lambda x.t:A\to B}\ (\to \text{right})$$

$$\frac{\Gamma,\vdash s:A \quad \Gamma, x:A\vdash t:B}{\Gamma\vdash t[x:=s]:B}\ (\text{cut})$$

FIGURE 10. λL-calculus

First we show the known relation between derivation in N and L: for all Γ and A the following holds

$$\Gamma \vdash_N A \Longleftrightarrow \Gamma \vdash_L A.$$

The following result was observed for N and λN by Curry, Howard, de Bruijn and Lambek. It is referred to as the Curry–Howard, formulae-as-types, proofs-as-terms and proofs-as-programs correspondence (interpretation, paradigm).

Theorem (Curry–Howard correspondence)**.** *Let S be one of the logical systems N, L or L^{cf} and let λS be the corresponding type assignment system. Then*

$$\Gamma \smallsetminus \mathbf{x} \vdash_S A \iff \exists t \in \Lambda^\circ(\mathbf{x})\ \Gamma \vdash_{\lambda S} t : A.$$

where $\Lambda^\circ(\mathbf{x}) = \{t \in \Lambda \mid Fv(t) \subseteq \mathbf{x}\}$.

The proof of the equivalence between systems N and L can be 'lifted' to that of λN and λL, i.e.,

$$\Gamma \vdash_{\lambda L} t : A \Longleftrightarrow \Gamma \vdash_{\lambda N} t : A.$$

Finally, using the cut-free system we get as bonus the Hauptsatz of [69] for minimal implicational sequent calculus, i.e.,

$$\Gamma \vdash_L A \Longleftrightarrow \Gamma \vdash_{L^{\mathrm{cf}}} A.$$

The main contribution of this work is expository, since it deals with well known results. In this work, the emphasis is on λ-terms rather than on derivations, since λ-terms are easier to reason about than two dimensional derivations.

7.2. **Logical interpretation of intersection types.** In Ghilezan [70] we consider the inhabitation in intersection and union type assignment system versus provability in intuitionistic (Heyting's) natural deduction propositional logic NJ with conjunction and disjunction (as given in Section 1.1, where the language of NJ contains also the constant $\top$).

$$\frac{\Gamma, x : A \vdash t_1 : C \quad \Gamma, x : B \vdash t_1 : C \quad \Gamma \vdash t_2 : A \cup B}{\Gamma \vdash t_1[x := t_2] : C}\ (\cup E)$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : A \cup B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash t : A \cup B}\ (\cup I)$$

FIGURE 11. $\lambda \cap \cup$: intersection and union type assignment system

The intersection and union type assignment system $\lambda\cap\cup$ is obtained by extending the system $\lambda\cap^\Omega$ with the rules given in Figure 11 where a pre-order $\leqslant$ on $\lambda \cap \cup$ is the extension of the pre-order $\leqslant$ on $\lambda\cap^\Omega$ obtained by adding the following rules: (i) $A \leqslant A \cup B, B \leqslant A \cup B$, (ii) $A \cup A \leqslant A$, (iii) $A \leqslant C, B \leqslant C \Rightarrow A \cup B \leqslant C$, and (iv) $A \cap (B \cup C) \leqslant (A \cap B) \cup (A \cap C)$.

The Curry–Howard correspondence between types inhabited in the intersection and union type assignment system and formulae provable in intuitionistic propositional logic with implication, conjunction, disjunction, and truth does not hold.

Inhabitation implies provability, but there are provable formulae which are not inhabited. This is shown in Hindley [95] in a syntactical way. We give a semantical proof of this fact by giving the appropriate type interpretations in $\mathcal{P}(D)$, starting from any lambda model $\mathcal{M} = \langle D, \cdot, [\![\ ]\!] \rangle$ (see [11]) and by mapping the set of intersection and union types into the set of propositional formulae that replaces each occurrence of $\cap$, $\cup$, and Ω in a type by $\wedge$, $\vee$, and $\top$ respectively.

The fact that types inhabited in $\lambda\cap^{\Omega}$ do not correspond to the provable formulae in intuitionistic propositional logic with $\rightarrow$ and $\wedge$, was shown in Hindley [95] by showing that the type

$$(A \rightarrow A) \cap ((A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C)$$

is not inhabited in $\lambda\cap^{\Omega}$ although it is provable in intuitionistic logic.

To show that some provable formulae are not inhabited we construct a model of $\lambda\cap^{\Omega}$ which is not a model of some provable formula, i.e., its interpretation in this model is empty.

In order to obtain the Curry–Howard correspondence for intuitionistic propositional logic LJ with conjunction and disjunction, we consider the extension of the simply typed lambda calculus with conjunction and disjunction types and the corresponding elimination and introduction rules, given in Figure 12. For this purpose, the set Type of types is given by the following

$$A, B = X \mid A \rightarrow B \mid A \wedge B \mid A \vee B$$

and the set of lambda terms Λ_c is obtained by expanding the set Λ with new constants c, c_1, and c_2 for conjunction and d, d_1, and d_2 for disjunction. $\lambda^{\rightarrow}_{\wedge}$ denotes the type assignment system obtained from $\lambda \rightarrow$ by adding the rules considering conjunction and λ_c denotes the type assignment system obtained from $\lambda \rightarrow$ by adding the rules considering conjunction and disjunction.

$$\frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash c_1 t : A} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash c_2 t : B}\ (\wedge E)$$

$$\frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash c t_1 t_2 : A \wedge B}\ (\wedge I)$$

$$\frac{\Gamma, x : A \vdash t_1 : C \quad \Gamma, x : B \vdash t_2 : C \quad \Gamma \vdash t_3 : A \cup B}{\Gamma \vdash d x t_1 t_2 t_3 : C}\ (\vee E)$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash d_1 t : A \vee B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash d_2 t : A \vee B}\ (\vee I)$$

FIGURE 12. λ_c: type assignment system with conjunction and disjunction

We link the inhabitation in the intersection and union type assignment system with the inhabitation in this extension of the simply typed lambda calculus. We

prove that inhabitation is decidable in $\lambda^{\rightarrow}_{\wedge}$ and λ_c by linking them to the question of decidability of provability in logics.

The difference between the special conjunction $\cap$ (called intersection) and the arbitrary propositional conjunction $\wedge$ is in the rule $(\cap I)$. In order to show that the term t has the intersection type it is necessary to show that t has both types in the same basis. t is the same in the conclusion as in both premises of the rule $(\cap I)$. The same holds for the rule $(\cap E)$. Thus in these two steps t remains the same although the deduction grows and the λ-terms do not correspond to the deductions. With the usual propositional conjunction $\wedge$ the lambda terms correspond to the deductions since it is possible to obtain a term of conjunction type from two terms with different types. Something similar happens with the special disjunction $\cup$ (called union).

In Dezani, Ghilezan and Venneri [45] we consider intersection and union types in Combinatory logic, which is a formal system equivalent to λ-calculus. In [45] we investigate the Curry–Howard correspondence between Hilbert (axiomatic) style intuitionistic logic and Combinatory logic. We propose a typed version of Combinatory logic with intersection and union types. This was a novelty, since all the existing systems with intersection types up to 1990s were type assignment systems. For the difference between typed systems (typeability *à la Church*) and type assignment systems (typeability *à la Curry*) we refer the reader to Barendregt [9]. Different typed lambda calculi with intersection types were further proposed by Liquori and Ronchi Della Rocca [23] and Bono et al. [108].

7.3. **Intersection types and topologies in λ-calculus.** In Ghilezan [78] typeability of terms in the full intersection type assignment system $\lambda\cap^{\Omega}$ is used to introduce topologies on the set of lambda terms Λ. We consider sets of lambda terms that can be typeable by a given type in a given environment:

$$\mathcal{V}_{\Gamma,A} = \{t \in \Lambda \mid \Gamma \vdash t : A\}.$$

For a fixed Γ the family of sets $\{\mathcal{V}_{\Gamma,A}\}_{A\in\mathsf{Type}}$ forms the basis of a topology on Λ, called the Γ-*fit topology*. Open sets in the Γ-fit topology are unions of basic open sets.

These topologies lead to simple proofs of some fundamental results of the lambda calculus such as the continuity theorem and the genericity lemma. We show that application is continuous, unsolvable terms are bottoms, and $\beta\eta$-normal forms are isolated points with respect to these topologies.

The restriction of these topologies to the set of closed lambda terms Λ°, called the *fit topology*, appears to be unique. It is defined by considering the set of all closed lambda terms that can be typed by a given type:

$$\mathcal{V}_{A} = \{t \in \Lambda^{\circ} \vdash t : A\}.$$

The family $\{\mathcal{V}_{A}\}_{A\in\mathsf{Type}}$ forms a basis for a topology on Λ°.

We compare the fit topology and the filter topology [11] and show that: (i) they coincide on the set Λ° of closed λ-terms, (ii) for every Γ-fit topology on the set Λ there is a coincident topology on Λ and vice versa.

The fit topology is a simpler description of the filter topology since the main difference between these topologies is that the former is a topology introduced on the set of types and then traced on terms by the inverse map, whereas the latter is introduced directly on the set of terms.

7.4. **Reducibility method.** The *reducibility method* is a well known framework for proving reduction properties of λ-terms typeable in different type systems. It was introduced by Tait [140] for proving the strong normalisation property of simply typed λ-calculus. Later it was used to prove *strong normalisation property* of various type systems in [141, 88, 101, 73], *the Church-Rosser property* (confluence) of $\beta\eta$-reduction in [100, 137, 111, 112] and to characterise some special classes of λ-terms such as strongly normalising terms, normalising terms, head normalising terms, and weak head normalising terms by their typeability in various intersection type systems in [67, 47, 41].

In Ghilezan and Likavec [86] we develop a general reducibility method for proving reduction properties of λ-terms typeable in intersection type systems with and without the universal type Ω, whereas in [87] we focus only on the intersection type assignment system $\lambda\cap^{\Omega}$ with the type Ω. Sufficient conditions for its application are derived. This method leads to uniform proofs of confluence, standardization, and weak head normalisation of terms typeable in the system with the type Ω. In this system the reducibility method can be extended to a proof method suitable to prove reduction properties of untyped λ-terms with certain invariance.

The general idea of the reducibility method is to provide a link between terms typeable in a type system and terms satisfying certain reduction properties (e.g., strong normalisation, confluence). For that reason types are interpreted by suitable sets of λ-terms: saturated and stable sets in Tait [140] and Krivine [101] and admissible relations in Mitchell [111] and [112]. These interpretations are based on the sets of terms considered (e.g., strong normalisation, confluence). Then the soundness of type assignment with respect to these interpretations is obtained. A consequence of soundness is that every term typeable in the type system belongs to the interpretation of its type. This is an intermediate step between the terms typeable in a type system and terms satisfying the reduction property considered.

Necessary notions for the reducibility method are (as presented in [87]): 1. type interpretation; 2. term valuations; 3. closure conditions; 4. soundness of the type assignment.

1. **Type interpretation**. We consider the set of all λ-terms Λ as the *applicative structure* whose domain are λ-terms and where the application is the application of terms. If $\mathcal{P} \subseteq \Lambda$ is a fixed set, the type interpretation $[\![-]\!] : \mathsf{Type} \to 2^{\Lambda}$ is defined by:

($I1$) $[\![X]\!] = \mathcal{P}$, X is an atom;
($I2$) $[\![A \cap B]\!] = [\![A]\!] \cap [\![B]\!]$;
($I3$) $[\![A \to B]\!] = ([\![A]\!] \Rightarrow [\![B]\!]) \cap \mathcal{P} = \{t \in \mathcal{P} \mid \forall s \in [\![A]\!] \quad ts \in [\![B]\!]\}$;
($I4$) $[\![\Omega]\!] = \Lambda$.

An important property of the type interpretation is that $[\![A]\!] \subseteq \mathcal{P}$ for all types $A \not\sim \Omega$.

2. **Term valuations**. Let $\rho : \mathtt{var} \to \Lambda$ be a valuation of term variables in Λ. Then $[\![-]\!]_\rho : \Lambda \to \Lambda$ is defined as follows

$$[\![t]\!]_\rho = t[x_1 := \rho(x_1), \ldots, x_n := \rho(x_n)], \text{ where } Fv(t) = \{x_1, \ldots, x_n\}.$$

The *semantic satisfiability relation* $\vDash$ connects the type interpretation with the term valuation.

(i) $\rho \vDash t : A$ iff $[\![t]\!]_\rho \in [\![A]\!]$;
(ii) $\rho \vDash \Gamma$ iff $(\forall (x : A) \in \Gamma)\ \rho(x) \in [\![A]\!]$;
(iii) $\Gamma \vDash t : A$ iff $(\forall \rho \vDash \Gamma)\ \rho \vDash t : A$.

3. **Closure conditions**. Let us impose some conditions on $\mathcal{P} \subseteq \Lambda$.

– $\mathcal{X} \subseteq \Lambda$ satisfies the $\mathcal{P}$-*variable property*, notation $VAR(\mathcal{P}, \mathcal{X})$, if

$$(\forall x \in \mathtt{var})\ (\forall n \geqslant 0)\ (\forall t_1, \ldots, t_n \in \mathcal{P})\ \ xt_1 \ldots t_n \in \mathcal{X}.$$

– $\mathcal{X} \subseteq \Lambda$ is $\mathcal{P}$-*saturated*, notation $SAT(\mathcal{P}, \mathcal{X})$, if

$$(\forall t, s \in \Lambda)\ (\forall n \geqslant 0)\ (\forall t_1, \ldots, t_n \in \mathcal{P})$$
$$t[x := s]t_1 \ldots t_n \in \mathcal{X} \ \Rightarrow\ (\lambda x.t)st_1 \ldots t_n \in \mathcal{X}.$$

– $\mathcal{X} \subseteq \Lambda$ is $\mathcal{P}$-*closed*, notation $CLO(\mathcal{P}, \mathcal{X})$, if $t \in \mathcal{X} \Rightarrow \lambda x.t \in \mathcal{P}$.

The preorder on types is interpreted as the set theoretic inclusion. We prove the following realizability property, which is referred to as the *soundness property* or the *adequacy property*.

Theorem (Soundness of the type assignment)**.** *If $VAR(\mathcal{P}, \mathcal{P})$, $SAT(\mathcal{P}, \mathcal{P})$, and $CLO(\mathcal{P}, \mathcal{P})$ are satisfied, then $\Gamma \vdash t : A \Rightarrow \Gamma \vDash t : A$.*

An immediate consequence of soundness is the following statement.

Theorem (Reducibility method)**.** *If $VAR(\mathcal{P}, \mathcal{P})$, $SAT(\mathcal{P}, \mathcal{P})$, and $CLO(\mathcal{P}, \mathcal{P})$, then for all types $A \not\sim \Omega$ and $A \not\sim \Omega \to B$, where $B \not\sim \Omega$*

$$\Gamma \vdash t : A \Rightarrow t \in \mathcal{P}.$$

Proof method for Λ. To establish a proof method for untyped λ-terms it is necessary that a set $\mathcal{P} \subseteq \Lambda$ is invariant under abstraction, i.e.,

$$t \in \mathcal{P} \ \Leftrightarrow\ \lambda x.t \in \mathcal{P}. \qquad \square$$

If $\mathcal{P}$ is invariant under abstraction and satisfies $VAR(\mathcal{P}, \mathcal{P})$ and $SAT(\mathcal{P}, \mathcal{P})$, then $\mathcal{P} = \Lambda$. This method is applicable when:

– $\mathcal{P} = \mathcal{C} = \{t \in \Lambda \mid \beta\text{-reduction is confluent on } t\}$;
– $\mathcal{P} = \mathcal{S} = \{t \mid \text{every reduction of } t \text{ can be done in a standard way}\}$;
– $\mathcal{P} = \mathcal{WN} = \{t \mid t \text{ is weakly head normalising}\}$.

In [86] we distinguish the following two different kinds of type interpretation with respect to a given set $\mathcal{P} \subseteq \Lambda$.

(i) The type interpretation $[\![-]\!] : \mathsf{Type} \to 2^\Lambda$ is defined by:
  ($I1$) $[\![X]\!] = \mathcal{P}$, X is an atom;
  ($I2$) $[\![A \cap B]\!] = [\![A]\!] \cap [\![B]\!]$;
  ($I3$) $[\![A \to B]\!] = [\![A]\!] \Rightarrow [\![B]\!] = \{t \in \Lambda \mid \forall s \in [\![A]\!] \quad ts \in [\![B]\!]\}$.

(ii) The Ω-type interpretation $[\![-]\!]^\Omega : \mathsf{Type}^\Omega \to 2^\Lambda$ is defined by
- (Ω1) $[\![X]\!]^\Omega = \mathcal{P}$, X is an atom;
- (Ω2) $[\![A \cap B]\!]^\Omega = [\![A]\!]^\Omega \cap [\![B]\!]^\Omega$;
- (Ω3) $[\![A \to B]\!]^\Omega = [\![A]\!]^\Omega \Rightarrow_\Omega [\![B]\!]^\Omega = ([\![A]\!]^\Omega \Rightarrow [\![B]\!]^\Omega) \cap \mathcal{P} = \{t \in \mathcal{WN} \mid \forall s \in [\![A]\!]^\Omega \quad ts \in [\![B]\!]^\Omega\}$;
- (Ω4) $[\![\Omega]\!]^\Omega = \Lambda$.

Also, we distinguish two different closure conditions which a given set $\mathcal{P} \subseteq \Lambda$ has to satisfy. By combining different type interpretations with appropriate closure conditions on $\mathcal{P} \subseteq \Lambda$ we prove the soundness of the type assignment in both cases. In this way a method for proving properties of λ-terms typeable with intersection types is obtained.

Preliminary version of the work presented in [87, 86] is [84].

The problem of typability in a type system is whether there exists a type for a given term. The typability in the full intersection type assignment system $\lambda\cap^\Omega$ is trivial since there exists a universal type Ω which can be assigned to every term in this system.

But without the rule (Ω), the situation changes. In Likavec [105] we focus on typability of terms in the intersection type assignment systems without the type Ω. We show that all the strongly normalising terms are typable in these systems. They are the only terms typable in these systems. We also present detailed proofs for [86, 87].

7.5. **Behavioural inverse limit models.** In Dezani et al. [44] we construct two inverse limit λ-models which completely characterise sets of terms with similar computational behaviours:

Normalisation properties

(1) A term t *has a normal form*, $t \in \mathcal{N}$, if t reduces to a normal form.
(2) A term t *has a head normal form*, $t \in \mathcal{HN}$, if t reduces to a term of the form $\lambda\vec{x}.y\vec{t}$ (where possibly y appears in $\vec{x}$).
(3) A term t *has a weak head normal form*, $t \in \mathcal{WN}$, if t reduces to an abstraction or to a term starting with a free variable.

Persistent normalisation properties

(1) A term t is *persistently normalising*, $t \in \mathcal{PN}$, if $t\vec{u} \in \mathcal{N}$ for all $\vec{u} \in \mathcal{N}$.
(2) A term t is *persistently head normalising*, $t \in \mathcal{PHN}$, if $t\vec{u} \in \mathcal{HN}$ for all $\vec{u} \in \Lambda$.
(3) A term t is *persistently weak head normalising*, $t \in \mathcal{PWN}$, if $t\vec{u} \in \mathcal{WN}$ for all $\vec{u} \in \Lambda$.

Closability properties

(1) A term t is *closable*, $t \in \mathcal{C}$, if t reduces to a closed term.
(2) A term t is *closable normalising*, $t \in \mathcal{CN}$, if t reduces to a closed normal form.
(3) A term t is *closable head normalising*, $t \in \mathcal{CHN}$, if t reduces to a closed head normal form.

We build two *inverse limit λ-models* $\mathcal{D}_\infty$ and $\mathcal{E}_\infty$, according to Scott [132], which completely characterise each of the mentioned sets of terms. For that we need to discuss the functional behaviours of the terms belonging to these classes with respect to the step functions. Given compact elements a and b in the Scott domains $\mathcal{A}$ and $\mathcal{B}$ respectively, the *step function* $\mathsf{a} \Rightarrow \mathsf{b}$ is defined by $\boldsymbol{\lambda}\mathsf{c}.$ **if** $\mathsf{a} \sqsubseteq \mathsf{c}$ **then** b **else** $\bot$

Definition of models

(1) Let $\mathcal{D}_\infty$ be the inverse limit λ-model obtained by taking as $\mathcal{D}_0$ the lattice in Figure 13, as $\mathcal{D}_1$ the lattice $[\mathcal{D}_0 \to \mathcal{D}_0]_\perp$, and by defining the embedding $\mathsf{i}_0^{\mathcal{D}} : \mathcal{D}_0 \to [\mathcal{D}_0 \to \mathcal{D}_0]_\perp$ as follows:

$$\mathsf{i}_0^{\mathcal{D}}(\hat{\mathsf{n}}) = (\bot \Rightarrow \hat{\mathsf{h}}) \sqcup (\mathsf{n} \Rightarrow \hat{\mathsf{n}}), \qquad \mathsf{i}_0^{\mathcal{D}}(\mathsf{n}) = (\hat{\mathsf{h}} \Rightarrow \mathsf{h}) \sqcup (\hat{\mathsf{n}} \Rightarrow \mathsf{n}),$$
$$\mathsf{i}_0^{\mathcal{D}}(\hat{\mathsf{h}}) = \bot \Rightarrow \hat{\mathsf{h}}, \qquad \mathsf{i}_0^{\mathcal{D}}(\mathsf{h}) = \hat{\mathsf{h}} \Rightarrow \mathsf{h}, \qquad \mathsf{i}_0^{\mathcal{D}}(\bot) = \bot.$$

(2) Let $\mathcal{E}_\infty$ be the inverse limit λ-model obtained by taking as $\mathcal{E}_0$ the cpo in Figure 13, as $\mathcal{E}_1$ the cpo $[\mathcal{E}_0 \to \mathcal{E}_0]$, and by defining the embedding $\mathsf{i}_0^{\mathcal{E}} : \mathcal{E}_0 \to [\mathcal{E}_0 \to \mathcal{E}_0]$ as follows:

$$\mathsf{i}_0^{\mathcal{E}}(\hat{\mathsf{n}}) = (\bot \Rightarrow \hat{\mathsf{h}}) \sqcup (\mathsf{n} \Rightarrow \hat{\mathsf{n}}), \qquad \mathsf{i}_0^{\mathcal{E}}(\mathsf{n}) = (\hat{\mathsf{h}} \Rightarrow \mathsf{h}) \sqcup (\hat{\mathsf{n}} \Rightarrow \mathsf{n}),$$
$$\mathsf{i}_0^{\mathcal{E}}(\hat{\mathsf{h}}) = \bot \Rightarrow \hat{\mathsf{h}}, \qquad \mathsf{i}_0^{\mathcal{E}}(\mathsf{h}) = \hat{\mathsf{h}} \Rightarrow \mathsf{h},$$
$$\mathsf{i}_0^{\mathcal{E}}(\mathsf{c}) = \mathsf{c} \Rightarrow \mathsf{c}, \qquad \mathsf{i}_0^{\mathcal{E}}(\bot) = \bot \Rightarrow \bot.$$

(3) We will denote the partial orders on $\mathcal{D}_\infty$ and $\mathcal{E}_\infty$ by $\sqsubseteq^{\mathcal{D}}$ and $\sqsubseteq^{\mathcal{E}}$, respectively.

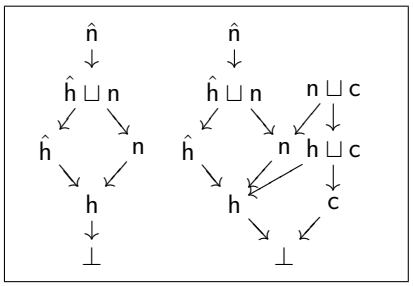

FIGURE 13. The lattice $\mathcal{D}_0$ and the cpo $\mathcal{E}_0$

More precisely, for each of these sets of terms there is a corresponding element in at least one of the two models such that a term belongs to the set if and only if its interpretation (in a suitable environment) is greater than or equal to that element. This is the result of the following theorem.

Theorem (Main Theorem, Version I)**.** *Let $\mathcal{D}_\infty$ and $\mathcal{E}_\infty$ be the inverse limit λ-models defined above and $\rho_{\hat{\mathsf{n}}}$ the environment defined by $\rho_{\hat{\mathsf{n}}}(x) = \hat{\mathsf{n}}$ for all $x \in \mathtt{var}$ (since each variable is in $\mathcal{PN}$). Then:*

(1) *$t \in \mathcal{PN}$ iff $[\![t]\!]^{\mathcal{D}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{D}} \hat{\mathsf{n}}$ iff $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \hat{\mathsf{n}}$;*
(2) *$t \in \mathcal{N}$ iff $[\![t]\!]^{\mathcal{D}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{D}} \mathsf{n}$ iff $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \mathsf{n}$;*
(3) *$t \in \mathcal{PHN}$ iff $[\![t]\!]^{\mathcal{D}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{D}} \hat{\mathsf{h}}$ iff $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \hat{\mathsf{h}}$;*

(4) $t \in \mathcal{HN}$ *iff* $[\![t]\!]^{\mathcal{D}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{D}} \mathsf{h}$ *iff* $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \mathsf{h}$;
(5) $t \in \mathcal{PWN}$ *iff* $[\![t]\!]^{\mathcal{D}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{D}} \bigsqcup_{n\in\mathbb{N}}(\underbrace{\bot \Rightarrow \ldots \Rightarrow \bot}_{n} \Rightarrow \bot)$;
(6) $t \in \mathcal{WN}$ *iff* $[\![t]\!]^{\mathcal{D}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{D}} \bot \Rightarrow \bot$;
(7) $t \in \mathcal{CN}$ *iff* $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \mathsf{c} \sqcup \mathsf{n}$;
(8) $t \in \mathcal{CHN}$ *iff* $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \mathsf{c} \sqcup \mathsf{h}$;
(9) $t \in \mathcal{C}$ *iff* $[\![t]\!]^{\mathcal{E}_\infty}_{\rho_{\hat{\mathsf{n}}}} \sqsupseteq^{\mathcal{E}} \mathsf{c}$.

This is proved by using the finitary logical descriptions of the models $\mathcal{D}_\infty$ and $\mathcal{E}_\infty$, obtained by defining two *intersection type assignment systems* in the following way. Starting from atomic types corresponding to the elements of $\mathcal{D}_0$ and $\mathcal{E}_0$, we construct the sets $\mathbb{T}^{\mathcal{D}}$ and $\mathbb{T}^{\mathcal{E}}$ of types using the *function type* constructor $\to$ and the *intersection type* constructor $\cap$ between *compatible* types, where two types are compatible if the corresponding elements have a join. Types are denoted by $A, B, A_1, \ldots$. $A^n \to B$ is short for $\underbrace{A \to \cdots \to A}_{n} \to B$ $(n \geqslant 0)$. The preorder between types is induced by reversing the order in the initial cpo and by encoding the initial embedding, according to the correspondence: (i) function type constructor corresponds to step function and (ii) intersection type constructor corresponds to join.

Then, we define the sets $\mathcal{F}^{\mathcal{D}}$ and $\mathcal{F}^{\mathcal{E}}$ of filters on the sets $\mathbb{T}^{\mathcal{D}}$ and $\mathbb{T}^{\mathcal{E}}$, respectively. Both $\mathcal{F}^{\mathcal{D}}$ and $\mathcal{F}^{\mathcal{E}}$, ordered by subset inclusion, are Scott domains. The compact elements are precisely the principal filters, and the bottom element is $\uparrow \Omega$. $\mathcal{F}^{\mathcal{D}}$ is an ω-algebraic complete lattice, since it has the top element $\mathbb{T}^{\mathcal{D}}$.

We can show that $\mathcal{F}^{\mathcal{D}}$ and $\mathcal{D}_\infty$ are isomorphic as ω-algebraic complete lattices, and that $\mathcal{F}^{\mathcal{E}}$ and $\mathcal{E}_\infty$ are isomorphic as Scott domains. This isomorphism falls in the general framework of *Stone dualities*. The interest of the above isomorphism lies in the fact that the interpretations of λ-terms in $\mathcal{D}_\infty$ and $\mathcal{E}_\infty$ are isomorphic to the filters of types one can derive in the corresponding type assignment systems. This gives the desired finitary logical descriptions of the models.

Theorem (Finitary logical descriptions)**.**

(1) *For any* $t \in \Lambda$ *and* $\rho : \mathtt{var} \mapsto \mathcal{F}^{\mathcal{D}}$, $[\![t]\!]^{\mathcal{F}^{\mathcal{D}}}_{\rho} = \{A \in \mathbb{T}^{\mathcal{D}} \mid \exists \Gamma.\, \Gamma \rhd \rho \,\&\, \Gamma \vdash^{\mathcal{D}} t : A\}$;
(2) *For any* $t \in \Lambda$ *and* $\rho : \mathtt{var} \mapsto \mathcal{F}^{\mathcal{E}}$, $[\![t]\!]^{\mathcal{F}^{\mathcal{E}}}_{\rho} = \{A \in \mathbb{T}^{\mathcal{E}} \mid \exists \Gamma.\, \Gamma \rhd \rho \,\&\, \Gamma \vdash^{\mathcal{E}} t : A\}$,
where $\Gamma \rhd \rho$ *means that for* $(x : B) \in \Gamma$ *one has that* $B \in \rho(x)$.

Therefore, the primary complete characterisation can be stated equivalently as follows: a term belongs to one of the nine sets mentioned if and only if it has a certain type (in a suitable basis) in one of the obtained type assignment systems. This is the result of the following theorem.

Theorem (Main Theorem, Version II)**.**

(1) $t \in \mathcal{PN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{D}} t : \hat{\nu}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \hat{\nu}$;
(2) $t \in \mathcal{N}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{D}} t : \nu$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \nu$;
(3) $t \in \mathcal{PHN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{D}} t : \hat{\mu}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \hat{\mu}$;
(4) $t \in \mathcal{HN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{D}} t : \mu$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \mu$;

(5) $t \in \mathcal{PWN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{D}} t : \Omega^n \to \Omega$ *for all* $n \in \mathbb{N}$;
(6) $t \in \mathcal{WN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{D}} t : \Omega \to \Omega$;
(7) $t \in \mathcal{CN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \gamma \cap \nu$;
(8) $t \in \mathcal{CHN}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \gamma \cap \mu$;
(9) $t \in \mathcal{C}$ *iff* $\Gamma_{\hat{\nu}} \vdash^{\mathcal{E}} t : \gamma$.

The proofs of the ($\Rightarrow$) parts are mainly straightforward inductions and case split, with the exception of the case of persistently normalising terms, which are treated using the notions of safe and unsafe suberms (see [21]). The proofs of the ($\Leftarrow$) parts require the set-theoretic semantics of intersection types and saturated sets, which is referred to as the reducibility method. To that purpose we define the *interpretations of types* in $\mathbb{T}^{\mathcal{D}}$ and in $\mathbb{T}^{\mathcal{E}}$ as follows:

Interpretation of types

(1) The map $[\![-]\!]^{\mathcal{D}} : \mathbb{T}^{\mathcal{D}} \to \mathcal{P}(\Lambda)$ is defined by:
  (i) $[\![\nu]\!]^{\mathcal{D}} = \mathcal{N}$, $[\![\hat{\nu}]\!]^{\mathcal{D}} = \mathcal{PN}$, $[\![\mu]\!]^{\mathcal{D}} = \mathcal{HN}$, $[\![\hat{\mu}]\!]^{\mathcal{D}} = \mathcal{PHN}$, $[\![\Omega]\!]^{\mathcal{D}} = \Lambda$;
  (ii) $[\![A \cap B]\!]^{\mathcal{D}} = [\![A]\!]^{\mathcal{D}} \cap [\![B]\!]^{\mathcal{D}}$;
  (iii) $[\![A \to B]\!]^{\mathcal{D}} = [\![A]\!]^{\mathcal{D}} \overset{\mathcal{D}}{\longmapsto} [\![B]\!]^{\mathcal{D}} = \{t \in \mathcal{WN} \mid \forall u \in [\![A]\!]^{\mathcal{D}} \quad tu \in [\![B]\!]^{\mathcal{D}}\}$.
(2) The map $[\![-]\!]^{\mathcal{E}} : \mathbb{T}^{\mathcal{E}} \to \mathcal{P}(\Lambda)$ is defined by:
  (i) $[\![\nu]\!]^{\mathcal{E}} = \mathcal{N}$, $[\![\hat{\nu}]\!]^{\mathcal{E}} = \mathcal{PN}$, $[\![\mu]\!]^{\mathcal{E}} = \mathcal{HN}$ $[\![\hat{\mu}]\!]^{\mathcal{E}} = \mathcal{PHN}$, $[\![\gamma]\!]^{\mathcal{E}} = \mathcal{C}$, $[\![\Omega]\!]^{\mathcal{E}} = \Lambda$;
  (ii) $[\![A \cap B]\!]^{\mathcal{E}} = [\![A]\!]^{\mathcal{E}} \cap [\![B]\!]^{\mathcal{E}}$;
  (iii) $[\![A \to B]\!]^{\mathcal{E}} = [\![A]\!]^{\mathcal{E}} \overset{\mathcal{E}}{\longmapsto} [\![B]\!]^{\mathcal{E}} = \{t \in \Lambda \mid \forall u \in [\![A]\!]^{\mathcal{E}} \quad tu \in [\![B]\!]^{\mathcal{E}}\}$.

The main contribution of the present paper is to show that *only two* models can characterise many different sets of terms. On the one hand it seems that we cannot find elements representing weak head normalisability and closability in the same model, since the first property requires the lifting of the space of functions and this does not agree with the second one. On the other hand, there are properties which appear strongly connected, like each normalisation property with its persistent version. It is not clear if these properties can be characterised separately, i.e., if one can build models in which only one of these properties is characterised.

A preliminary version of the present paper (dealing only with the first six sets of terms) is [41]. An extended abstract of the present paper is [43].

8. Intuitionistic sequent calculus and λ^{Gtz}-calculus

8.1. **Intersection types for λ^{Gtz}-calculus.** In Espírito Santo et al. [57], we introduce intersection types for the λ^{Gtz}-calculus. The set Type of types, ranged over by $A, B, C, \ldots, A_1, \ldots$, is defined inductively:

$$A, B \; ;::= \; X \mid A \to B \mid A \cap B$$

where X ranges over a denumerable set $TVar$ of type atoms.

The type assignment system $\lambda^{\mathsf{Gtz}}\cap$ is given in Figure 14.

The following rules are admissible in $\lambda^{\mathsf{Gtz}}\cap$:

1. If $\Gamma, x : A_i \vdash t : C$ then $\Gamma, x : \cap A_i \vdash t : C$.
2. If $\Gamma, x : A_i; D \vdash k : C$ then $\Gamma, x : \cap A_i; D \vdash k : C$.

$$\frac{}{\Gamma, x : \cap A_i \vdash x : A_i \;\; i = 1, \ldots, n, \;\; n \geqslant 1} \; (Ax)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \; (\to_R) \qquad \frac{\Gamma \vdash u : A_i \;\; \forall i \quad \Gamma; B \vdash k : C}{\Gamma; \cap A_i \to B \vdash u :: k : C} \; (\to_L)$$

$$\frac{\Gamma \vdash t : A_i, \;\; \forall i \quad \Gamma; \cap A_i \vdash k : B}{\Gamma \vdash tk : B} \; (Cut) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma; A \vdash \widehat{x}.t : B} \; (Sel)$$

FIGURE 14. $\lambda^{\mathsf{Gtz}}\cap$: type assignment system for λ^{Gtz}-calculus

Basis expansion and bases intersection are defined in an obvious way. Standard form of generation lemma holds for λ^{Gtz}.

Example: In λ-calculus with intersection types, the term $\lambda x.xx$ has the type $(A \cap (A \to B)) \to B$. The corresponding term in λ^{Gtz}-calculus is $\lambda x.x(x :: \widehat{y}.y)$. Although being a normal form this term is not typeable in the simply typed λ^{Gtz}-calculus. It is typeable in $\lambda^{\mathsf{Gtz}}\cap$ in the following way:

$$\dfrac{\dfrac{\dfrac{}{x : A \cap (A \to B) \vdash x : A \to B}\,(Ax) \qquad \dfrac{\dfrac{}{x : A \cap (A \to B) \vdash x : A}\,(Ax) \qquad \dfrac{\dfrac{}{x : A \cap (A \to B), y : B \vdash y : B}\,(Ax)}{x : A \cap (A \to B); B \vdash \widehat{y}.y : B}\,(Sel)}{x : A \cap (A \to B); A \to B \vdash (x :: \widehat{y}.y) : B}\,(\to_L)}{x : A \cap (A \to B) \vdash x(x :: \widehat{y}.y) : B}\,(Cut)}{\vdash \lambda x.x(x :: \widehat{y}.y) : (A \cap (A \to B)) \to B}\,(\to_R).$$

8.2. **Subject reduction and strong normalisation.** Basic properties of this system are analysed and the Subject reduction property is proved i.e.,

$$\text{If } \Gamma \vdash t : A \text{ and } t \to t', \text{ then } \Gamma \vdash t' : A.$$

The reduction μ is of different nature, since it reduces contexts instead of terms. A similar result for this reduction rule is given, i.e.,

$$\text{If } \Gamma; \cap B_i \vdash \widehat{x}.xk : A, \text{ then } \Gamma; B_i \vdash k : A, \text{ for some } i.$$

In [81], a slightly modified type assignment system $\lambda^{\mathsf{Gtz}}\cap$ with respect to the one given in 8.1 is considered. Subject reduction holds for this system as well.

We use intersection types in [57] to give a characterisation of the strongly normalising terms of an intuitionistic sequent calculus (where LJ easily embeds). The sequent term calculus presented in this paper integrates smoothly the λ-terms with generalised application or explicit substitution.

In order to prove that typeability in $\lambda^{\mathsf{Gtz}}\cap$ implies strong normalisation for the $\lambda^{\mathsf{Gtz}}\cap$, we connect it with the well-known system $\mathcal{D}$ for the λ-calculus (given in Section 2.2) via an appropriate mapping, and then use strong normalisation theorem for λ-terms typeable in system $\mathcal{D}$.

Terms in $\mathcal{D}$ are ordinary λ-terms equipped with the following two reduction relations, in addition to standard β reduction:

$$(\pi_1) \quad (\lambda x.M)NP \to (\lambda x.MP)N \qquad (\pi_2) \quad M((\lambda x.P)N) \to (\lambda x.MP)N.$$

We let $\pi = \pi_1 \cup \pi_2$. We use capital letters here to denote the terms in $\mathcal{D}$ to differentiate them from the terms in $\lambda^{\mathsf{Gtz}}\cap$.

We define a mapping F from λ^{Gtz} to λ. The idea is the following. If $F(t) = M$, $F(u_i) = N_i$ and $F(v) = P$, then $t(u_1 :: u_2 :: (x)v)$, say, is mapped to $(\lambda x.P)(MN_1N_2)$. Formally, a mapping $F : \lambda^{\mathsf{Gtz}}Terms \longrightarrow \lambda Terms$ is defined simultaneously with an auxiliary mapping $F' : \lambda Terms \times \lambda^{\mathsf{Gtz}}Contexts \longrightarrow \lambda Terms$ as follows:

$$\begin{array}{ll} F(x) = x & F'(N, \widehat{x}.t) = (\lambda x.F(t))N \\ F(\lambda x.t) = \lambda x.F(t) & F'(N, u :: k) = F'(NF(u), k). \\ F(tk) = F'(F(t), k) & \end{array}$$

We prove the following theorems:

- **Soundness of F:** If $\lambda^{\mathsf{Gtz}}\cap$ proves $\Gamma \vdash t : A$, then $\mathcal{D}$ proves $\Gamma \vdash F(t) : A$.
- **Reduction of SN:** For all $t \in \lambda^{\mathsf{Gtz}}$, if $F(t)$ is $\beta\pi$-SN, then t is $\beta\pi\sigma\mu$-SN.

The main theorem is the following:

Theorem (Typeability $\Rightarrow$ SN)**.** *If a λ^{Gtz}-term t is typeable in $\lambda^{\mathsf{Gtz}}\cap$, then t is $\beta\pi\sigma\mu$-SN.*

In order to prove that SN implies typeability we prove the following:

- $\beta\pi\sigma$-normal forms and $\beta\pi\sigma\mu$-normal forms of the λ^{Gtz}-calculus are typeable in the $\lambda^{\mathsf{Gtz}}\cap$ system.
- **Subject expansion property:** If $t \to t'$, t is the redex and t' is typeable in $\lambda^{\mathsf{Gtz}}\cap$, then t is typeable in $\lambda^{\mathsf{Gtz}}\cap$.

The main theorem is the following:

Theorem (SN $\Rightarrow$ typeability)**.** *All strongly normalising ($\beta\sigma\pi\mu - SN$) terms are typeable in the $\lambda^{\mathsf{Gtz}}\cap$ system.*

Finally, in order to deal with generalised applications and explicit substitutions, we consider two extensions of the λ-calculus: the ΛJ-calculus, where application $M(N, x.P)$ is *generalised* [98]; and the $\lambda\mathtt{x}$-calculus, where substitution $M[x := N]$ is *explicit* [128]. Intersection types have been used to characterise the strongly normalising terms of both ΛJ-calculus [109] and $\lambda\mathtt{x}$-calculus [104]. But in both [109] and [104] the "natural" typing rules for generalised application or substitution had to be suplemented with extra rules in order to secure that every strongly normalising term is typeable. Hence, the "natural" rules failed to capture the strongly normalising terms. We prove that λ^{Gtz} and $\lambda^{\mathsf{Gtz}}\cap$ are useful for resolving these issues.

Let t be a λ^{Gtz}-term.

(1) t is a *λJ-term* if every cut occurring in t is of the form $t(u :: \widehat{x}.v)$.
(2) t is a *$\lambda\mathtt{x}$-term* if every cut occurring in t has one of the forms $t(u :: \widehat{x}.x)$ or $t(\widehat{x}.v)$.

We define appropriate type assignment systems $\lambda J\cap$ and $\lambda\mathbf{x}\cap$ for these calculi and prove the following:

(1) Let t be a λJ-term. t is $\beta\pi\sigma\mu - SN$ iff t is typeable in $\lambda J\cap$.
(2) Let t be a $\lambda\mathbf{x}$-term. t is $\beta\pi\sigma\mu - SN$ iff t is typeable in $\lambda\mathbf{x}\cap$.

9. Classical natural deduction and $\lambda\mu$-calculus

9.1. **Terms for natural deduction and sequent calculus classical logic.** In Ghilezan [80], the work of Barendregt and Ghilezan [12] is further elaborated and its results are generalised for classical logic. Two extensionally equivalent type assignment systems for the $\lambda\mu$-calculus are considered. The type assignment system $\lambda\mu N$ is actually the simply typed $\lambda\mu$-calculus, given in Figure 8. It corresponds to implicational fragment of classical natural deduction NK (given in Figure 2), whereas the type assignment system $\lambda\mu L$ given in Figure 15 corresponds to implicational fragment of classical sequent calculus LK (given in Figure 4). In addition, a cut free variant of $\lambda\mu L$, denoted by $\lambda\mu L^{\mathrm{cf}}$, is introduced and used to give a short proof of Cut elimination theorem for classical logic.

$$\frac{}{\Gamma, y : A \vdash y : A, \Delta}\ (\text{axiom})$$

$$\frac{\Gamma \vdash u : A, \Delta \quad \Gamma, x : B \vdash t : C, \Delta}{\Gamma, y : A \to B \vdash t[x := yu] : C, \Delta}\ (\to \text{left}) \qquad \frac{\Gamma, y : A \vdash t : B, \Delta}{\Gamma \vdash \lambda y.t : A \to B, \Delta}\ (\to \text{right})$$

$$\frac{\Gamma \vdash t : A, \Delta, \beta : A, \alpha : B}{\Gamma \vdash \mu\alpha.[\beta]t : B, \Delta, \beta : A}\ (\mu)$$

$$\frac{\Gamma \vdash u : B, \Delta \quad \Gamma, x : B \vdash t : A, \Delta}{\Gamma \vdash t[x := u] : A, \Delta}\ (\text{cut})$$

FIGURE 15. $\lambda\mu L$-calculus

In Figure 15 a *term context* $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ is a set of variable declarations such that for every variable x_i there is at most one declaration $x_i : A_i$ in Γ and a *co-term context* $\Delta = \{\alpha_1 : B_1, \ldots, \alpha_k : B_k\}$ is a set of co-variable declarations such that for every co-variable α_l there is at most one declaration $\alpha_l : B_l$ in Δ. In this setup $\Gamma \setminus \mathbf{x} = \{A_1, \ldots A_n\}$ and $\Delta \setminus \alpha = \{B_1, \ldots B_k\}$.

It is shown that the statement A is derivable from assumptions in Γ in NK if and only if it is derivable from the same assumptions in LK, i.e., for all Γ and A

$$\Gamma \vdash_{NK} A, \Delta \Longleftrightarrow \Gamma \vdash_{LK} A, \Delta.$$

The following result was given by Parigot [117] as an extension of the well-known proposition-as-types interpretation of intuitionistic logic.

Theorem (Curry–Howard correspondence for classical logic)**.** *If* SK *is one of the logical systems* NK*,* LK *or* $\mathsf{LK}^{\mathsf{cf}}$ *and if* $\lambda\mu S$ *is the corresponding type assignment system, then*

$$\Gamma \smallsetminus \mathbf{x} \vdash_{SK} A, \Delta \smallsetminus \alpha \iff \exists t \in \Lambda^{\circ}(\mathbf{x}) \cup \Lambda^{\circ}_{\mu}(\alpha)\ \Gamma \vdash_{\lambda\mu S} t : A, \Delta.$$

where $\Lambda^{\circ}(\vec{x}) = \{t \in \Lambda \mid Fv(t) \subseteq \mathbf{x}\}$, $\Lambda^{\circ}_{\mu}(\alpha) = \{t \in \Lambda \mid Fv_{\mu}(t) \subseteq \alpha\}$

It is also proved that

$$\Gamma \vdash_{\lambda\mu L} t : C, \Delta \Longleftrightarrow \Gamma \vdash_{\lambda\mu N} t : C, \Delta.$$

Finally, using the type assignment system $\lambda\mu L^{\mathrm{cf}}$, Cut elimination theorem of Gentzen [69] for classical implicational sequent calculus is proved, i.e.,

$$\Gamma \vdash_{LK} A \Longleftrightarrow \Gamma \vdash_{LK^{\mathrm{cf}}} A.$$

The type assignment system $\lambda\mu L$ is a novel system for encoding proofs in classical sequent logic. The main focus of this paper is on $\lambda\mu$-terms, rather than on derivations.

9.2. **Separability in $\lambda\mu$-calculus.** In Herbelin and Ghilezan [94], we investigate the separability property of $\lambda\mu$-calculus. In the untyped λ-calculus Böhm's theorem deals with the separability property of λ-terms [20, 35, 10, 101]. For two different normal forms there is a context such that one of these terms converges in this context, whereas the other one diverges in the same context. A consequence of this theorem is that $\beta\eta$ equality is the *maximal consistent equality* between λ-terms having normal forms. Hence, if t and u are two λ–terms having different $\beta\eta$ normal forms, meaning that $t = u$ cannot be proved in λ-calculus, and if this calculus is extended with $t = u$, then according to Böhm's theorem every equality of λ-terms can be proved in the extended calculus. In other words such an extended calculus is inconsistent.

Two terms are *observationally equivalent* if, whenever put in the same context, either they both make it reducible to a normal form or they both make it diverge. More generally, two terms may be considered as equivalent if, when observed from outside, they exhibit the same behaviour. Therefore another important consequence of Böhm's separability in the λ-calculus setting is that observational equivalence for normalisable terms coincides with $\beta\eta$-equivalence. The proof of Böhm's theorem can be considered as a refutation procedure for observational equivalence. An overview of the relation between Böhm's theorem and observational equivalence is given by Dezani-Ciancaglini and Giovannetti [46].

Regarding computational interpretations of classical logic Böhm's separability property has been investigated in Parigot's $\lambda\mu$-calculus, so far. David and Py [38] showed that Parigot's $\lambda\mu$-calculus does not satisfy Böhm's separability property. This means that the equality of Parigot's $\lambda\mu$-calculus is not the maximal consistent equality between $\lambda\mu$-terms having normal forms.

Saurin [131] studied the Böhm's separability property in a syntactic modification of the $\lambda\mu$-calculus by de Groote [39] which is denoted here by $\Lambda\mu$ following Saurin.

The syntax of the $\Lambda\mu$-calculus is given by the following:

$$t \quad ::= \quad x \mid \lambda x.t \mid tu \mid \mu\beta.t \mid [\alpha]t.$$

The reduction rules are the same as of the $\lambda\mu$-calculus (see Subsection 4).

Saurin showed that the $\Lambda\mu$-calculus is a strict extension of Parigot's $\lambda\mu$-calculus and that it enjoys Böhm's separability property. Therefore, the equality in $\Lambda\mu$-calculus is the maximal consistent equality of $\lambda\mu$-terms having normal forms. The two syntax were up to now considered as almost the same. Obviously this subtle move in the syntax has significant consequences. In [94] we restored Böhm separability in $\lambda\mu$ by extending the syntax of $\lambda\mu$ with a dynamically bound continuation variable $\widehat{\mathsf{tp}}$ and the reduction rules with two rules

$$\begin{array}{rcl} [\widehat{\mathsf{tp}}]\mu\widehat{\mathsf{tp}}.c & \to & c \\ \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t & \to & t. \end{array}$$

In this way we obtained $\lambda\mu\widehat{\mathsf{tp}}$, actually its call-by-name variant. It is possible to establish then a mutual embedding of $\Lambda\mu$ and $\lambda\mu\widehat{\mathsf{tp}}$. Embedding of $\Lambda\mu$ into the extended $\lambda\mu$, actually $\Pi : \Lambda\mu \longrightarrow \lambda\mu\widehat{\mathsf{tp}}$ is given by the following:

$$\begin{array}{lcl} \Pi(x) & \triangleq & x \\ \Pi(\lambda x.t) & \triangleq & \lambda x.\Pi(t) \\ \Pi(t\,s) & \triangleq & \Pi(t)\,\Pi(s) \\ \Pi(\mu\alpha.t) & \triangleq & \mu\alpha.[\widehat{\mathsf{tp}}]\Pi(t) \\ \Pi([\alpha]t) & \triangleq & \mu\widehat{\mathsf{tp}}.[\alpha]\Pi(t). \end{array}$$

Embedding of the extended $\lambda\mu$ into $\Lambda\mu$, actually $\Sigma : \lambda\mu\widehat{\mathsf{tp}} \longrightarrow \Lambda\mu$ is the following:

$$\begin{array}{lcll} \Sigma(x) & \triangleq & x & \\ \Sigma(\lambda x.t) & \triangleq & \lambda x.\Sigma(t) & \\ \Sigma(t\,s) & \triangleq & \Sigma(t)\,\Sigma(s) & \\ \Sigma(\mu\alpha.[\beta]t) & \triangleq & \mu\alpha.([\beta]\Sigma(t)) & \text{if } \beta \text{ and } \widehat{\mathsf{tp}} \text{ are distinct} \\ \Sigma(\mu\alpha.[\widehat{\mathsf{tp}}]t) & \triangleq & \mu\alpha.(\Sigma(t)) & \\ \Sigma(\mu\widehat{\mathsf{tp}}.[\alpha]t) & \triangleq & [\alpha]\Sigma(t) & \text{if } \alpha \text{ and } \widehat{\mathsf{tp}} \text{ are distinct} \\ \Sigma(\mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]t) & \triangleq & \Sigma(t). & \end{array}$$

From the desired properties:

- $t = u$ in $\Lambda\mu$ implies $\Pi(t) = \Pi(u)$ in $\lambda\mu\widehat{\mathsf{tp}}$,
- $t = u$ in $\lambda\mu\widehat{\mathsf{tp}}$ implies $\Sigma(t) = \Sigma(u)$ in $\Lambda\mu$,

we conclude the separability of the extended $\lambda\mu$-calculus.

Theorem (Separability)**.** *$\lambda\mu\widehat{tp}$, the extended $\lambda\mu$-calculus is observationally complete for normal forms, i.e., for any two normal forms there exists an evaluation $\lambda\mu\widehat{tp}$-context $C[]$, such that, in $\lambda\mu\widehat{tp}$, $C[t] = x$ and $C[s] = y$ for x and y being arbitrary fresh variables.*

Separability in simply typed $\lambda\mu$-calculus is an open question. It was shown in [136, 134, 53] that separability in simply typed λ-calculus needs different treatment from Böhm's method for the untyped λ-calculus. There is ongoing research along the lines of the approach by Došen and Petrić [53].

$$
\begin{array}{rcl}
X \in \mathit{TypeConstants} & & \\
A, B & ::= & X \mid A_\Sigma \to B \\
\Gamma & ::= & \emptyset \mid \Gamma, x : A_\Sigma \\
\Delta & ::= & \emptyset \mid \Delta, \alpha : A \\
\Sigma, \Xi & ::= & \bot \mid A \cdot \Sigma
\end{array}
$$

$$\frac{}{\Gamma, x : A_\Sigma \vdash_\Sigma x : A; \Delta}\ Ax$$

$$\frac{\Gamma, x : A_\Sigma \vdash_\Xi t : B; \Delta}{\Gamma \vdash_\Xi \lambda x.t : (A_\Sigma \to B); \Delta}\ (\to_i) \qquad \frac{\Gamma \vdash_\Xi t : (A_\Sigma \to B); \Delta \quad \Gamma \vdash_\Sigma s : A; \Delta}{\Gamma \vdash_\Xi t\, s : B; \Delta}\ (\to_e)$$

$$\frac{\Gamma \vdash_\Sigma c : \bot; \Delta, \alpha : A}{\Gamma \vdash_\Sigma \mu\alpha.c : A; \Delta} \qquad \frac{\Gamma \vdash_{A\cdot\Sigma} c : \bot; \Delta}{\Gamma \vdash_\Sigma \mu\widehat{\mathsf{tp}}.c : A; \Delta} \qquad \frac{\Gamma \vdash_\Sigma t : A; \Delta, \alpha : A}{\Gamma \vdash_\Sigma [\alpha]t : \bot; \Delta, \alpha : A} \qquad \frac{\Gamma \vdash_\Sigma t : A; \Delta}{\Gamma \vdash_{A\cdot\Sigma} [\widehat{\mathsf{tp}}]t : \bot; \Delta}$$

FIGURE 16. Simple typing of $\lambda\mu\widehat{\mathsf{tp}}$-calculus

9.3. **Simple types for extended $\lambda\mu$-calculus.** In Herbelin and Ghilezan [94] we propose a system of simple types for call-by-name $\lambda\mu\widehat{\mathsf{tp}}$, the $\lambda\mu$-calculus extended by a dynamically bound continuation variable, which is introduced in the previous subsection. Like for typing $\lambda\mu$, we have two kinds of sequents, one for each category of expressions:

$$\begin{array}{ll} \Gamma \vdash_\Sigma t : A; \Delta & \text{(for terms)} \\ \Gamma \vdash_\Sigma c : \bot; \Delta & \text{(for commands).} \end{array}$$

Like for $\lambda\mu$, we have a context of *hypotheses* Γ that assigns types to term variables and a context of *conclusions* Δ that assigns types to continuation variables. But we have also to take care of the $\mu\widehat{\mathsf{tp}}$ dynamic binder.

There is an extra data to type the dynamic effects. Each use of $\mu\widehat{\mathsf{tp}}$ pushes the current continuation on a stack of dynamically bound continuations. Each call to $\widehat{\mathsf{tp}}$ pops the top continuation from this stack. The extra information needed to type the dynamic binding is not a single formula but the ordered list Σ of the types of the continuations present in the stack.

The type system, given in Figure 16 enjoys preservation of types under reduction.

Theorem (Subject reduction)**.**

(i) *If* $\Gamma \vdash_\Sigma t : A; \Delta$ *and* $t \to s$*, then* $\Gamma \vdash_\Sigma s : A; \Delta$.
(ii) *If* $\Gamma \vdash_\Sigma c : \bot; \Delta$ *and* $c \to c'$*, then* $\Gamma \vdash_\Sigma c' : \bot; \Delta$.

10. Classical sequent calculus and $\overline{\lambda}\mu\widetilde{\mu}$-calculus

10.1. **Confluence of call-by-name and call-by-value disciplines.** In Likavec and Lescanne [107], we deal with untyped $\overline{\lambda}\mu\widetilde{\mu}$-calculus and its semantics, with complete proofs given in [106].

This work investigates some properties of $\overline{\lambda}\mu\widetilde{\mu}_T$ and $\overline{\lambda}\mu\widetilde{\mu}_Q$, the two subcalculi of untyped $\overline{\lambda}\mu\widetilde{\mu}$-calculus of Curien and Herbelin [34], closed under the call-by-name

and the call-by-value reduction, respectively. The syntax and reduction rules of $\overline{\lambda}\mu\widetilde{\mu}$ were given in Section 5.

First of all, the proof of confluence for both versions of the $\overline{\lambda}\mu\widetilde{\mu}$-calculus is given, adopting the method of parallel reductions given by Takahashi [142]. This approach consists of simultaneously reducing all the redexes existing in a term.

We present the proof for $\overline{\lambda}\mu\widetilde{\mu}_T$, the proof for $\overline{\lambda}\mu\widetilde{\mu}_Q$ being a straightforward modification of the proof for $\overline{\lambda}\mu\widetilde{\mu}_T$. The complete proofs can be found in [106]. We denote the reduction defined by the three reduction rules for $\overline{\lambda}\mu\widetilde{\mu}_T$ by $\rightarrow_{\mathrm{n}}$ and its reflexive, transitive, and closure by congruence by $\twoheadrightarrow_{\mathrm{n}}$.

First, we define the notion of parallel reduction $\Rightarrow_{\mathrm{n}}$ for $\overline{\lambda}\mu\widetilde{\mu}_T$. We prove that $\twoheadrightarrow_{\mathrm{n}}$ is reflexive and transitive closure of $\Rightarrow_{\mathrm{n}}$, so in order to prove the confluence of $\twoheadrightarrow_{\mathrm{n}}$, it is enough to prove the diamond property for $\Rightarrow_{\mathrm{n}}$. The diamond property for $\Rightarrow_{\mathrm{n}}$, follows from the stronger "Star property" for $\Rightarrow_{\mathrm{n}}$ that we prove.

The parallel reduction, denoted by $\Rightarrow_{\mathrm{n}}$ is defined inductively, as follows:

$$\frac{}{x \Rightarrow_{\mathrm{n}} x}\ (g1_n) \qquad \frac{v \Rightarrow_{\mathrm{n}} v'}{\lambda x\,.\,t \Rightarrow_{\mathrm{n}} \lambda x\,.\,t'}\ (g2_n) \qquad \frac{c \Rightarrow_{\mathrm{n}} c'}{\mu\alpha\,.\,c \Rightarrow_{\mathrm{n}} \mu\alpha\,.\,c'}\ (g3_n)$$

$$\frac{}{\alpha \Rightarrow_{\mathrm{n}} \alpha}\ (g4_n) \qquad \frac{v \Rightarrow_{\mathrm{n}} v', E \Rightarrow_{\mathrm{n}} E'}{v \bullet E \Rightarrow_{\mathrm{n}} v' \bullet E'}\ (g5_n) \qquad \frac{c \Rightarrow_{\mathrm{n}} c'}{\widetilde{\mu}x\,.\,c \Rightarrow_{\mathrm{n}} \widetilde{\mu}x\,.\,c'}\ (g6_n)$$

$$\frac{v \Rightarrow_{\mathrm{n}} v', e \Rightarrow_{\mathrm{n}} e'}{\langle v \parallel e\rangle \Rightarrow_{\mathrm{n}} \langle v' \parallel e'\rangle}\ (g7_n) \qquad \frac{v_1 \Rightarrow_{\mathrm{n}} v_1', v_2 \Rightarrow_{\mathrm{n}} v_2', E \Rightarrow_{\mathrm{n}} E'}{\langle \lambda x\,.\,t_1 \parallel v_2 \bullet E\rangle \Rightarrow_{\mathrm{n}} \langle v_1'[x := v_2'] \parallel E'\rangle}\ (g8_n)$$

$$\frac{c \Rightarrow_{\mathrm{n}} c', E \Rightarrow_{\mathrm{n}} E'}{\langle \mu\alpha\,.\,c \parallel E\rangle \Rightarrow_{\mathrm{n}} c'[\alpha := E']}\ (g9_n) \qquad \frac{v \Rightarrow_{\mathrm{n}} v', c \Rightarrow_{\mathrm{n}} c'}{\langle v \parallel \widetilde{\mu}x\,.\,c\rangle \Rightarrow_{\mathrm{n}} c'[x := v']}\ (g10_n).$$

It is easy to prove that for every term G:

1. $G \Rightarrow_{\mathrm{n}} G$;
2. If $G \rightarrow_{\mathrm{n}} G'$ then $G \Rightarrow_{\mathrm{n}} G'$;
3. If $G \Rightarrow_{\mathrm{n}} G'$ then $G \twoheadrightarrow_{\mathrm{n}} G'$;
4. If $G \Rightarrow_{\mathrm{n}} G'$ and $H \Rightarrow_{\mathrm{n}} H'$,
   then $G[x := H] \Rightarrow_{\mathrm{n}} G'[x := H']$ and $G[\alpha := H] \Rightarrow_{\mathrm{n}} G'[\alpha := H']$.

From 2. and 3. we conclude that $\twoheadrightarrow_{\mathrm{n}}$ is the reflexive and transitive closure of $\Rightarrow_{\mathrm{n}}$.

Next, we define the term G^* which is obtained from G by simultaneously reducing all the existing redexes of the term G.

$(*1_n)$ $x^* \equiv x$ $\quad$ $(*2_n)$ $(\lambda x\,.\,t)^* \equiv \lambda x\,.\,t^*$ $\quad$ $(*3_n)$ $(\mu\alpha\,.\,c)^* \equiv \mu\alpha\,.\,c^*$

$(*4_n)$ $\alpha^* \equiv \alpha$ $\quad$ $(*5_n)$ $(v \bullet E)^* \equiv v^* \bullet E^*$ $\quad$ $(*6_n)$ $(\widetilde{\mu}x\,.\,c)^* \equiv \widetilde{\mu}x\,.\,c^*$

$(*7_n)$ $(\langle v \parallel e\rangle)^* \equiv \langle v^* \parallel e^*\rangle$ if $\langle v \parallel e\rangle \neq \langle \lambda x\,.\,t_1 \parallel v_2 \bullet E\rangle$,
$\langle v \parallel e\rangle \neq \langle \mu\alpha\,.\,c \parallel E\rangle$ and $\langle v \parallel e\rangle \neq \langle v \parallel \widetilde{\mu}x\,.\,c\rangle$

$(*8_n)$ $(\langle \lambda x\,.\,t_1 \parallel v_2 \bullet E\rangle)^* \equiv \langle v_1^*[x := v_2^*] \parallel E^*\rangle$

$(*9_n)$ $(\langle \mu\alpha\,.\,c \parallel E\rangle)^* \equiv c^*[\alpha := E^*]$

$(*10_n)$ $(\langle v \parallel \widetilde{\mu}x\,.\,c\rangle)^* \equiv c^*[x := v^*]$

We prove that if $G \Rightarrow_{\mathrm{n}} G'$ then $G' \Rightarrow_{\mathrm{n}} G^*$. Then it is easy to deduce the diamond property for $\Rightarrow_{\mathrm{n}}$: if $G_1\ {}_{\mathrm{n}}\!\Leftarrow G \Rightarrow_{\mathrm{n}} G_2$ then $G_1 \Rightarrow_{\mathrm{n}} G'\ {}_{\mathrm{n}}\!\Leftarrow G_2$ for some G'. Finally,

from the previous, it follows that $\overline{\lambda}\mu\widetilde{\mu}_T$ is confluent, i.e., if $G_1 \,{}_{\mathrm{n}}\!\!\twoheadleftarrow G \twoheadrightarrow_{\mathrm{n}} G_2$ then $G_1 \twoheadrightarrow_{\mathrm{n}} G' \,{}_{\mathrm{n}}\!\!\twoheadleftarrow G_2$ for some G'.

As a step towards a better understanding of denotational semantics of $\overline{\lambda}\mu\widetilde{\mu}$-calculus, its *untyped* call-by-value ($\overline{\lambda}\mu\widetilde{\mu}_Q$) and call-by-name ($\overline{\lambda}\mu\widetilde{\mu}_T$) versions are interpreted. Untyped $\overline{\lambda}\mu\widetilde{\mu}$-calculus is Turing-complete, hence a naive set-theoretic approach would not be enough. Continuation semantics of $\overline{\lambda}\mu\widetilde{\mu}_Q$ and $\overline{\lambda}\mu\widetilde{\mu}_T$ is given using the category of negated domains of [138], and Moggi's Kleisli category over predomains for the continuation monad [113]. Soundness theorems are given for both, call-by-value and call-by-name subcalculi, thus relating operational and denotational semantics. A detailed account on the literature on continuation semantics is also given. Lack of space forbids us to give a detailed account on the semantics here.

10.2. **Strong normalisation in unrestricted $\overline{\lambda}\mu\widetilde{\mu}$-calculus.** In Dougherty et al. [51], we develop a new intersection type system for the $\overline{\lambda}\mu\widetilde{\mu}$-calculus of Curien and Herbelin [34]. The system in this work improves on earlier type disciplines for $\overline{\lambda}\mu\widetilde{\mu}$ (including the current authors' [48, 49]): in addition to characterising the $\overline{\lambda}\mu\widetilde{\mu}$ expressions that are strongly normalising under free (unrestricted) reduction, the system enjoys the Subject reduction and the Subject expansion properties.

The set Type of *raw types* is generated from an infinite set $TVar$ of type-variables as follows

$$A, B ::= TVar \mid A \to B \mid A^\circ \mid A \cap B$$

where A° is the *dual* type of type A. We consider raw types modulo the equality generated by saying that (i) intersection is associative and commutative and (ii) for all raw types A, $A^{\circ\circ} = A$,

A *type* is either a *term-type* or a *coterm-type* or the special constant $\bot$. A raw type is a *term-type* if it is either a type variable, or of the form $(A_1 \to A_2)$ or $(A_1 \cap \cdots \cap A_k)$, $i \geqslant 2$ for term-types A_i, or of the form D° for a coterm-type D. A raw type is a *coterm-type* if it is either a coterm variable, or of the form A° for a term-type A or of the form $(D_1 \cap \cdots \cap D_k)$, $i \geqslant 2$ for coterm-types D_i. Note that every coterm-type is a type of the form A°, where A is a term-type, or an intersection of such types.

Each type other than $\bot$ is uniquely—up to the equivalences mentioned above—of one of the forms in the table below. Furthermore, for each type T there is a unique type which is T°. If T is a term-type [resp., coterm-type] then T° is a coterm-type [resp., term-type].

	term-types	coterm-types
	τ	τ
	$(A_1 \to A_2)$	$(A_1 \to A_2)^\circ$
for $n \geqslant 2$:	$(A_1 \cap A_2 \cap \cdots \cap A_n)$	$(A_1 \cap A_2 \cap \cdots \cap A_n)^\circ$
for $n \geqslant 2$:	$(A_1^\circ \cap A_2^\circ \cap \cdots \cap A_n^\circ)^\circ$	$(A_1^\circ \cap A_2^\circ \cap \cdots \cap A_n^\circ)$.

The characterisation of the two columns as being "term-types" or "coterm-types" holds under the convention that the A_i displayed are all term-types.

We refer to types of the form $(A \to B)$ and $(A_1 \cap \cdots \cap A_k) \to B$ uniformly using the notation $(\bigcap A_i \to B)$, with the understanding that the $\bigcap A_i$ might refer to a single non-intersection type.

The type assignment system $\mathcal{M}^\cap$is given by the typing rules in Figure 17, where v is any (co)variable.

$$\frac{}{\Sigma,\ \ v : (T_1 \cap \cdots \cap T_k) \ \vdash\ v : T_i}\ (\text{ax})$$

$$\frac{\Sigma,\ \ x : A \ \vdash\ r : B}{\Sigma \ \vdash\ \lambda x.r \ :\ A \to B}\ (\to \text{r}) \qquad \frac{\Sigma \ \vdash\ r : A_i \quad i = 1, \ldots, k \qquad \Sigma \ \vdash\ e : B^\circ}{\Sigma \ \vdash\ r \bullet e : ((A_1 \cap \cdots \cap A_k) \to B)^\circ}\ (\to \text{e})$$

$$\frac{\Sigma,\ \ \alpha : A^\circ \ \vdash\ c : \bot}{\Sigma \ \vdash\ \mu\alpha.c : A}\ (\mu) \qquad \frac{\Sigma,\ \ x : A \ \vdash\ c : \bot}{\Sigma \ \vdash\ \widetilde{\mu}x.c : A^\circ}\ (\widetilde{\mu})$$

$$\frac{\Sigma \ \vdash\ r : A \qquad \Sigma \ \vdash\ e : A^\circ}{\Sigma \ \vdash\ \langle r \parallel e \rangle : \bot}\ (\text{cut})$$

FIGURE 17. The typing system $\mathcal{M}^\cap$

In the system presented here there is no unrestricted $\cap$-introduction rule which is significant for the treatments of Subject reduction and Type soundness. Intersection types can be generated for redexes by the (μ) or $(\widetilde{\mu})$ rules only. The rationale behind the new type system is to accept the introduction of an intersection only at specific positions and specific times when typing an expression, namely when an arrow is introduced on the left; then a type intersection is only introduced at the parameter position. Still, the new system types exactly all the strongly normalising expressions.

Example: The normal form $\text{`}lx.\mu\alpha.\langle x \parallel x \bullet \alpha\rangle$, which corresponds to the normal form $\text{`}lx.xx$ in $\text{`}l$-calculus, is not typable in $\overline{\lambda}\mu\widetilde{\mu}$ with simple types. It is typable in the currently introduced system $\mathcal{M}^\cap$ by $\text{`}lx.\mu\alpha.\langle x \parallel x \bullet \alpha\rangle : A \cap (A \to B) \to B$.

Theorem (Subject expansion)**.** *Let t and s be arbitrary terms or coterms and let v be a variable or covariable. Suppose $\Sigma \ \vdash\ t[v := s] : T$ and suppose that s is typable in context Σ. Then there is a type $D = (D_1 \cap \cdots \cap D_k)$, $\quad k \geqslant 1$, such that*

$$\Sigma \ \vdash\ s : D_i \quad \textit{for each } i \qquad \textit{and} \qquad \Sigma, v : D \ \vdash\ t : T.$$

Theorem (Main result)**.** *A $\overline{\lambda}\mu\widetilde{\mu}$ term is strongly normalising if and only if it is typable in $\mathcal{M}^\cap$.*

It is straightforward to prove that strong normalisation implies typeability using the fact that normal forms are typeable.

To prove strong normalisation under free reduction for typable expressions is more challenging. The difficulty using a traditional reducibility (or "candidates")

argument arises from the critical pairs $\langle \mu`a.c \parallel \widetilde{\mu}x.d\rangle$. Since neither of the expressions here can be identified as the preferred redex one cannot define candidates by induction on the structure of types.

The "symmetric candidates" technique in [6, 121] uses a fixed-point technique to define the candidates and suffices to prove strong normalisation for simply-typed $\overline{\lambda}\mu\widetilde{\mu}$, but the interaction between intersection types and symmetric candidates is technically problematic.

In order to prove that typeable expressions are SN we first construct *pairs* (R, E) given by two non-empty sets $T \subseteq \Lambda_r$ and $C \subseteq \Lambda_e$. The pair (R, E) is *stable* if for every $r \in R$ and every $e \in E$, the command $\langle r \parallel e\rangle$ is SN. A pair (R, E) is *saturated* if

- whenever $\mu`a.c$ satisfies $\forall e \in E, c[\alpha := e]$ is SN then $\mu`a.c \in R$, and
- whenever $\widetilde{\mu}x.c$ satisfies $\forall r \in R, c[x := r]$ is SN then $\widetilde{\mu}x.c \in E$.

A pair (R, E) is *simple* if no term in R is of the form $\mu`a.c$ and no coterm in E is of the form $\widetilde{\mu}x.c$.

We show that if the original pair is stable and simple, then we may always construct the saturated, stable extension. To achieve this we define the maps: $\Phi_r : 2^{\Lambda_e} \to 2^{\Lambda_r}$ and $\Phi_e : 2^{\Lambda_r} \to 2^{\Lambda_e}$ by

$$\begin{aligned}\Phi_r(Y) = \{r \mid r \text{ is of the form } \mu\alpha.c \text{ and } \forall e \in Y, c[\alpha := e] \text{ is SN}\}\\ \cup \{r \mid r \text{ is simple and } \forall e \in Y, \langle r \parallel e\rangle \text{ is SN}\}\end{aligned}$$

$$\begin{aligned}\Phi_e(X) = \{e \mid e \text{ is of the form } \widetilde{\mu}x.c \text{ and } \forall r \in X, c[x := r] \text{ is SN}\}\\ \cup \{e \mid e \text{ is simple and } \forall r \in X, \langle r \parallel e\rangle \text{ is SN.}\}\end{aligned}$$

Since each of Φ_e and Φ_r is antimonotone, the maps $(\Phi_r \circ \Phi_e) : \Lambda_r \to \Lambda_r$ and $(\Phi_e \circ \Phi_r) : \Lambda_e \to \Lambda_e$ are monotone, so each of these maps has a complete lattice of fixed points, ordered by set inclusion.

We define different saturated pairs to interpret types depending on whether the type to be interpreted is (i) an arrow-type or its dual or (ii) an intersection or its dual.

If R is a simple set of SN terms let $R^{\uparrow}$ be the least fixed point of $(\Phi_r \circ \Phi_e)$ with the property that $R \subseteq R^{\uparrow}$. Analogously, $E^{\uparrow}$ is the least fixed point of $(\Phi_e \circ \Phi_r)$ such that $E \subseteq E^{\uparrow}$.

For interpreting the types that are intersections or their duals, we use the fact that the collection of fixed points of $(\Phi_r \circ \Phi_e)$ (and that of $(\Phi_e \circ \Phi_r)$) carries its own lattice structure under inclusion. We need the following definitions.

- Let $\mathsf{Fix}_{(\Phi_r \circ \Phi_e)}$ be the set of fixed points of the operator $(\Phi_r \circ \Phi_e)$. If $R_1, \ldots, R_k$ are fixed points of $(\Phi_r \circ \Phi_e)$, let $(R_1 \curlywedge \ldots \curlywedge R_k)$ denote the meet of these elements in the lattice $\mathsf{Fix}_{(\Phi_r \circ \Phi_e)}$.
- Let $\mathsf{Fix}_{(\Phi_e \circ \Phi_r)}$ be the set of fixed points of the operator $(\Phi_e \circ \Phi_r)$. Let $(E_1 \curlywedge \ldots \curlywedge E_k)$ denote the meet of fixed points of $(\Phi_e \circ \Phi_r)$.

Interpretation of types For each type T we define the set $[\![T]\!]$, maintaining the invariant that when T is a term-type then $[\![T]\!]$ is a fixed point of $(\Phi_r \circ \Phi_e)$ (set

of terms) and when T is a coterm-type then $[\![T]\!]$ is a fixed point of $(\Phi_e \circ \Phi_r)$ (set of coterms).

- When T is $\bot$ then $[\![T]\!]$ is the set of SN commands.
- When T is a type variable we set R to be the set of term variables, then construct the pair $(R^\uparrow, \Phi_e(R^\uparrow))$. We then take $[\![T]\!]$ to be $R^\uparrow$ and $[\![T^\circ]\!]$ to be $\Phi_e(R^\uparrow)$.
- Suppose T is $(\bigcap A_i \to B)$. Set E to be $\{r \bullet e \mid \forall i, r \in [\![A_i]\!] \text{ and } e \in [\![B^\circ]\!]\}$ then construct the pair $(\Phi_r(E^\uparrow), E^\uparrow)$. We then take $[\![T]\!]$ to be $\Phi_r(E^\uparrow)$ and $[\![T^\circ]\!]$ to be $(E^\uparrow)$.
- When T is $(A_1 \cap A_2 \cdots \cap A_n)$, $\quad n \geqslant 2$, we take $[\![T]\!]$ to be $([\![A_1]\!] \curlywedge \ldots \curlywedge [\![A_n]\!])$ and then take $[\![T^\circ]\!]$ to be $\Phi_e([\![T]\!])$.
- When T is $(A_1^\circ \cap A_2^\circ \cdots \cap A_n^\circ)^\circ$, $\quad n \geqslant 2$, we take $[\![T^\circ]\!]$ to be $([\![A_1^\circ]\!] \curlywedge \ldots \curlywedge [\![A_n^\circ]\!])$ and then take $[\![T]\!]$ to be $\Phi_r([\![T^\circ]\!])$.

The following collects the information we need to prove Type soundness.

(1)
(2) For each type T, $[\![T]\!]$ is a set of SN (co)terms. $[\![(\bigcap A_i \to B)^\circ]\!] \supseteq \{r \bullet e \mid \forall i, r \in [\![A_i]\!] \text{ and } e \in [\![B^\circ]\!]\}$.
(3) $(\lambda x.b) \in [\![(\bigcap A_i \to B)]\!]$ if for all r such that $\forall i, r \in [\![A_i]\!]$ we have $b[x := r] \in [\![B]\!]$.
(4) $(\mu\text{`}a.c) \in [\![A]\!]$ if for all $e \in [\![A^\circ]\!]$ we have $c[\text{`}a := e]$ SN. Similarly, $(\widetilde{\mu}x.c) \in [\![A^\circ]\!]$ if for all $r \in [\![A]\!]$ we have $c[x := r]$ SN.
(5) $[\![(T_1 \cap \cdots \cap T_k)]\!] \subseteq ([\![T_i]\!] \cap \cdots \cap [\![T_k]\!])$.

Theorem (Type soundness)**.** *If expression t is typable with type T then t is in $[\![T]\!]$.*

Since each $[\![T]\!]$ consists of SN expressions Type soundness implies that all typable expressions are SN.

General consideration of symmetry led us in [48, 49] to consider intersection *and union* types in symmetric λ-calculi. These papers characterised strong normalisation for call-by-name and call-by-value restrictions of the $\overline{\lambda}\mu\widetilde{\mu}$-calculus, whereas the results in this work apply to unrestricted reduction. We might argue that if a term has type $A \cap B$, meaning that it denotes values which inhabit both A and B, then it can interact with any continuation that can receive an A-value *or* a B-value: such a continuation will naturally be expected to have the type $A \cup B$. But any type that can be the type of a variable can be the type of a coterm (via the $\widetilde{\mu}$-construction) and any type that can be the type of a covariable can be the type of a term (via the μ-construction). This would suggest having intersections *and* unions for terms and continuations. It is well-known [119, 7] that the presence of union types causes difficulties for the Subject reduction property; unfortunately our attempt to recover Subject reduction in [48] was in error, as was pointed out to us by Hugo Herbelin [91]. Hence, this work only takes into account intersection types. The use of an explicit involution operator allows us to record the relationship between an intersection $(A \cap B)$ and its dual type $(A \cap B)^\circ$. The "classical" nature of the underlying logic is reflected in the "double-negation".

10.3. **Dual calculus.** Wadler's Dual calculus was introduced in [147, 148] as a term calculus which corresponds to classical sequent logic. In Dougherty et al. [52], we investigate some syntactic properties of Wadler's Dual calculus and establish some of the key properties of the underlying reduction.

We give now the syntax and reduction rules of Wadler's Dual calculus (although in our slightly altered notation). We distinguish three syntactic categories: *terms*, *coterms*, and *statements*. Terms yield values, while coterms consume values. A statement is a cut of a term against a coterm.

If r, q range over the set Λ_r of terms, e, f range over the set Λ_e of coterms, and c ranges over statements, then the syntax of the Dual calculus is given by the following:

$$\begin{array}{lrcl} \mathsf{Term:} & r, q & ::= & x \mid \langle r,\ q\rangle \mid \langle r\rangle\mathrm{inl} \mid \langle r\rangle\mathrm{inr} \mid [e]\mathrm{not} \mid \mu\alpha\,.\,c \\ \mathsf{Coterm:} & e, f & ::= & \alpha \mid [e,\ f] \mid \mathrm{fst}[e] \mid \mathrm{snd}[e] \mid \mathrm{not}\langle r\rangle \mid \widetilde{\mu}x\,.\,c \\ \mathsf{Command:} & c & ::= & (\!|\ r\ \bullet\ e\ |\!) \end{array}$$

where x ranges over a set of term variables Var_{R}, $\langle r,\ q\rangle$ is a pair, $\langle r\rangle\mathrm{inl}$ ($\langle r\rangle\mathrm{inr}$) is an injection on the left (right) of the sum, $[e]\mathrm{not}$ is a complement of a coterm, and $\mu\alpha\,.\,c$ is a covariable abstraction. Next, α ranges over a set of covariables Var_{L}, $[e,\ f]$ is a case, $\mathrm{fst}[e]$ ($\mathrm{snd}[e]$) is a projection from the left (right) of a product, $\mathrm{not}\langle r\rangle$ is a complement of a term, and $\widetilde{\mu}x\,.\,c$ is a variable abstraction. Finally $(\!|\ r\ \bullet\ e\ |\!)$ is a cut. The term variables can be bound by μ-abstraction, whereas the coterm variables can be bound by $\widetilde{\mu}$-abstraction. The sets of free term and coterm variables, Fv_{R} and Fv_{L}, are defined as usual, respecting Barendregt's convention [10] that no variable can be both, bound and free, in the expression.

The reduction rules for an unrestricted calculus are given in Figure 18.

$$\begin{array}{lrcl} (\beta\widetilde{\mu}) & (\!|\ r\ \bullet\ \widetilde{\mu}x\,.\,c\ |\!) & \to & c[x := r] \\ (\beta\mu) & (\!|\ \mu\alpha\,.\,c\ \bullet\ e\ |\!) & \to & c[\alpha := e] \\ (\beta\wedge) & (\!|\ \langle r,\ q\rangle\ \bullet\ \mathrm{fst}[e]\ |\!) & \to & (\!|\ r\ \bullet\ e\ |\!) \\ (\beta\wedge) & (\!|\ \langle r,\ q\rangle\ \bullet\ \mathrm{snd}[e]\ |\!) & \to & (\!|\ q\ \bullet\ e\ |\!) \\ (\beta\vee) & (\!|\ \langle r\rangle\mathrm{inl}\ \bullet\ [e,\ f]\ |\!) & \to & (\!|\ r\ \bullet\ e\ |\!) \\ (\beta\vee) & (\!|\ \langle r\rangle\mathrm{inr}\ \bullet\ [e,\ f]\ |\!) & \to & (\!|\ r\ \bullet\ f\ |\!) \\ (\beta\neg) & (\!|\ [e]\mathrm{not}\ \bullet\ \mathrm{not}\langle r\rangle\ |\!) & \to & (\!|\ r\ \bullet\ e\ |\!) \end{array}$$

FIGURE 18. Reduction rules for the Dual calculus

The basic system is not confluent, inheriting the well-known anomaly of classical cut-elimination. Wadler recovers confluence by restricting to reduction strategies corresponding to (either of) the call-by-value or call-by-name disciplines.

The two subcalculi $\mathsf{Dual}_{\mathrm{R}}$ and $\mathsf{Dual}_{\mathrm{L}}$ are obtained by giving the priority to $(\widetilde{\mu})$ redexes or to (μ) redexes, respectively. $\mathsf{Dual}_{\mathrm{R}}$ is defined by refining the reduction rule $(\beta\mu)$ as follows

$$(\!|\ \mu\alpha.c\ \bullet\ \underline{e}\ |\!) \to c[\alpha := \underline{e}] \qquad \text{provided } \underline{e} \text{ is a coterm not of the form } \widetilde{\mu}x.c'$$

and $\mathsf{Dual}_{\mathrm{L}}$ is defined similarly by refining the reduction rule $(\beta\widetilde{\mu})$ as follows

$$\langle\!| \underline{r} \bullet \widetilde{\mu}x \,.\, c |\!\rangle \to c[x := \underline{r}] \qquad \text{provided } \underline{r} \text{ is a term not of the form } \mu\text{`}a.c'.$$

We show that once the "critical pair" in the reduction system is removed by giving priority to either the "left" or to the "right" reductions, confluence holds in both the typed and untyped versions of the term calculus. Although the critical pair can be disambiguated in two ways, the proof we give dualises to yield confluence results for each system. The proof is an application of Takahashi's parallel reductions technique [142], analogous to the one used in [107] and with details of the proves given in [106].

A complementary perspective to that of considering the Dual calculus as term-assignment to logic proofs is that of viewing sequent proofs as typing derivations for raw expressions. The set Type of types corresponds to the logical connectives; for the Dual calculus the set of types is given by closing a set of *base types* X under conjunction, disjunction, and negation

$$A, B \;::=\; X \mid A \wedge B \mid A \vee B \mid \neg A.$$

Type bases have two components, the *antecedent* a set of bindings of the form $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, and the *succedent* of the form $\Delta = \alpha_1 : B_1, \ldots, \alpha_k : B_k$, where x_i, α_j are distinct for all $i = 1, \ldots, n$ and $j = 1, \ldots, k$. The judgements of the type system are given by the following:

$$\Gamma \vdash \Delta, \boxed{r : A} \qquad \boxed{e : A}, \Gamma \vdash \Delta \qquad c \;:\; (\Gamma \vdash \Delta)$$

where Γ is the antecedent and Δ is the succedent. The first judgement is the typing for a term, the second is the typing for a coterm and the third one is the typing for a statement. The box denotes a distinguished output or input, i.e., a place where the computation will continue or where it happened before. The type assignment system for the Dual calculus, introduced by Wadler [147, 148], is given in Figure 10.3.

We prove strong normalisation (SN) for unrestricted reduction of typed terms, including expansion rules capturing extensionality. The proof is a variation on the "semantical" method of reducibility, where types are interpreted as *pairs* of sets of terms. Our proof technique uses a fixed-point construction similar to that in [6] but the technique is considerably simplified.

The approach is similar to the one given for [51] so we just present the details that differ. The pairs are defined analogously, as well as the notion of stable, saturated, and simple pairs.

We can always expand a pair to be saturated. Also if the original pair is stable and simple, then we may always construct the saturated, stable extension.

We define the following constructions on pairs, where script letters denote pairs, and if $\mathcal{P}$ is a pair, $\mathcal{P}_{\mathrm{R}}$ and $\mathcal{P}_{\mathrm{L}}$ denote its component sets of terms and coterms.

Let $\mathcal{P}$ and $\mathcal{Q}$ be pairs.

- The pair $(\mathcal{P} \curlywedge \mathcal{Q})$ is given by:
  - $(\mathcal{P} \curlywedge \mathcal{Q})_{\mathrm{R}} = \{\langle r_1,\, r_2\rangle \mid r_1 \in \mathcal{P}_{\mathrm{R}},\, r_2 \in \mathcal{Q}_{\mathrm{R}}\}$
  - $(\mathcal{P} \curlywedge \mathcal{Q})_{\mathrm{L}} = \{\mathrm{fst}[e] \mid e \in \mathcal{P}_{\mathrm{L}}\} \cup \{\mathrm{snd}[e] \mid e \in \mathcal{Q}_{\mathrm{L}}\}$.

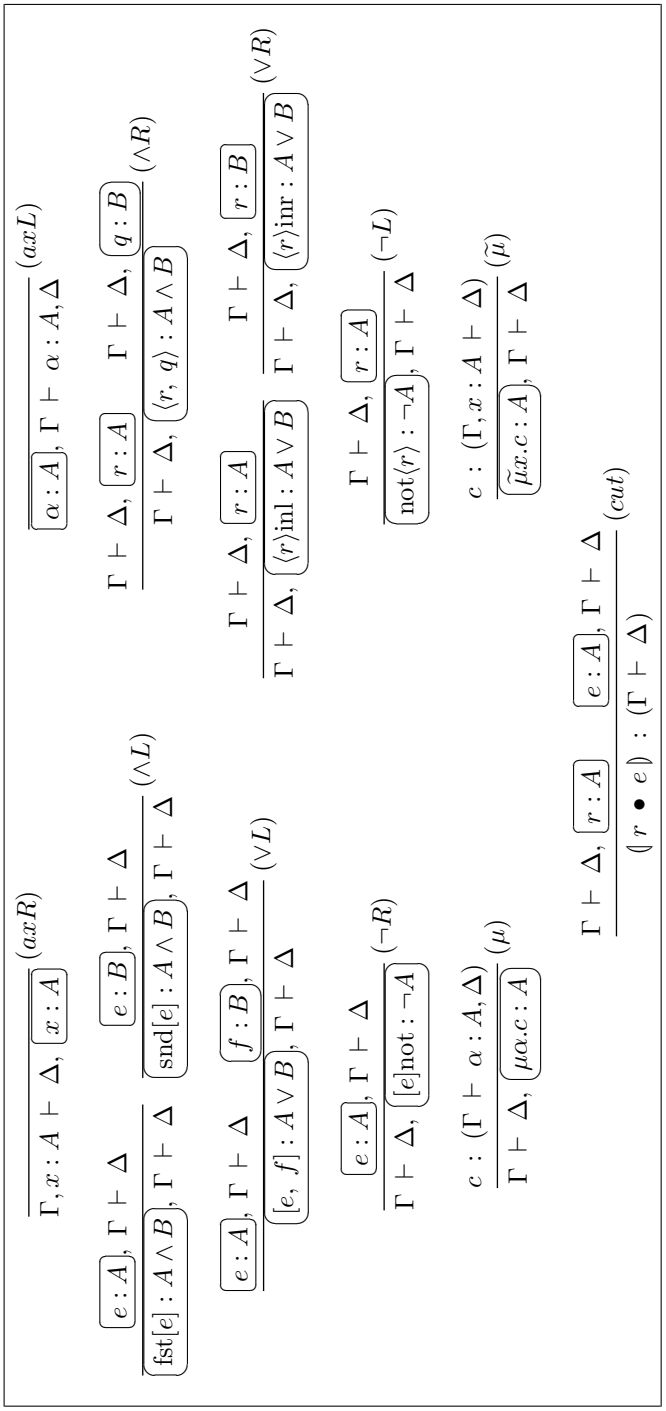

FIGURE 19. Type system for the Dual calculus

- The pair $(\mathcal{P} \curlyvee \mathcal{Q})$ is given by:
  - $(\mathcal{P} \curlyvee \mathcal{Q})_{\mathrm{R}} = \{\langle r\rangle\mathrm{inl} \mid r \in \mathcal{P}_{\mathrm{R}}\} \cup \{\langle r\rangle\mathrm{inr} \mid r \in \mathcal{Q}_{\mathrm{R}}\}$
  - $(\mathcal{P} \curlyvee \mathcal{Q})_{\mathrm{L}} = \{[e_1,\ e_2] \mid e_1 \in \mathcal{P}_{\mathrm{L}}.\ e_2 \in \mathcal{Q}_{\mathrm{L}}\}$.
- The pair $\mathcal{P}^{\circ}$ is given by:
  - $(\mathcal{P}^{\circ})_{\mathrm{R}} = \{[e]\mathrm{not} \mid e \in \mathcal{P}_{\mathrm{L}}\}$
  - $(\mathcal{P}^{\circ})_{\mathrm{L}} = \{\mathrm{not}\langle r\rangle \mid r \in \mathcal{P}_{\mathrm{R}}\}$.

Each of $(\mathcal{P} \curlywedge \mathcal{Q})$, $(\mathcal{P} \curlyvee \mathcal{Q})$, and $\mathcal{P}^{\circ}$ is simple and we show that if $\mathcal{P}$ and $\mathcal{Q}$ are stable pairs, then $(\mathcal{P} \curlywedge \mathcal{Q})$, $(\mathcal{P} \curlyvee \mathcal{Q})$, and $\mathcal{P}^{\circ}$ are each stable.

The type-indexed family of pairs $\mathcal{S} = \{\mathcal{S}^T \mid T \in \mathsf{Type}\}$ is defined as follows, which is our notion of reducibility candidates for the Dual calculus:

- When T is a base type, $\mathcal{S}^T$ is any stable saturated extension of $(\mathit{Var}_{\mathrm{R}}, \mathit{Var}_{\mathrm{L}})$.
- $\mathcal{S}^{A\wedge B}$ is any stable saturated extension of $(\mathcal{S}^A \curlywedge \mathcal{S}^B)$.
- $\mathcal{S}^{A\vee B}$ is any stable saturated extension of $(\mathcal{S}^A \curlyvee \mathcal{S}^B)$.
- $\mathcal{S}^{\neg A}$ is any stable saturated extension of $(\mathcal{S}^A)^{\circ}$.

Next we prove that typeable terms and coterms lie in the candidates $\mathcal{S}$, i.e., if term r is typeable with type A then r is in $\mathcal{S}^A_{\mathrm{R}}$ and if coterm e is typeable with type A then e is in $\mathcal{S}^A_{\mathrm{L}}$. Since $\mathcal{S}^A_{\mathrm{R}}$ and $\mathcal{S}^A_{\mathrm{L}}$ consist of SN expressions, it follows that typeable terms and coterms are SN. If $t = c$ is a typeable statement then it suffices to observe that, taking $`a$ to be any covariable not occurring in $c$, the term $\mu`a.c$ is typeable. This proves the strong normalisation of all typeable expressions of the calculus.

10.4. **Symmetric calculus.** Another interesting calculus expressing a computational interpretation of classical logic is the Symmetric Lambda Calculus of Barbanera and Berardi [6], which was originally used to extract the constructive content of classical proofs. In Dougherty et al. [50] we explore the use of intersection types for symmetric proof calculi. More specifically we characterise termination in the (propositional version of) the Symmetric Lambda Calculus of Barbanera and Berardi [6].

The syntax of λ^{sym} expressions is given by the following:

$$t := x \mid \langle t_1, t_2\rangle \mid \sigma_1(t), \mid \sigma_2(t), \mid \lambda x.c \mid (t_1\ *\ t_2).$$

We depart from [6] in that we treat the operator $*$ as syntactically commutative.

The reduction rules of the calculus are

$$(\lambda x.b\ *\ a) \to b[x := a] \qquad (\langle t_1, t_2\rangle\ *\ \sigma_i(u)) \to \langle t_i, u\rangle$$
$$\lambda x.(b\ *\ x) \to b \text{ if } x \text{ not free in } b.$$

The set Type of *raw types* is generated from an infinite set $TVar$ of type-variables as follows

$$A, B ::=\ TVar \mid A \wedge B \mid A \vee B \mid A^{\perp} \mid A \cap B.$$

We consider raw types modulo the equations

$$A^{\perp\perp} = A \qquad (A \wedge B)^{\perp} = A^{\perp} \vee B^{\perp} \qquad (A \vee B)^{\perp} = A^{\perp} \wedge B^{\perp}.$$

A *type* is either an equivalence class modulo these equations or the special type $\bot$. Note that by orienting the equations above left-to-right each type has a normal form, in which the $(\cdot)^\bot$ operator is applied only to type variables or intersections. It is then easy to see that each type other than $\bot$ is uniquely of one of the following forms (where τ is a type variable):

$$\tau \quad \tau^\bot \quad (A_1 \wedge \cdots \wedge A_n) \quad (A_1 \vee \cdots \vee A_n) \quad (A_1 \cap \cdots \cap A_n) \quad (A_1 \cap \cdots \cap A_n)^\bot.$$

The type assignment system $\mathcal{B}$ is given by the typing rules in Figure 20.

$$\frac{}{\Sigma,\ x : (T_1 \cap \cdots \cap T_k) \ \vdash\ x : T_i}\ (\text{ax})$$

$$\frac{\Sigma \ \vdash\ t_1 : A_1 \qquad \Sigma \ \vdash\ t_2 : A_2}{\Sigma \ \vdash\ \langle t_1, t_2\rangle : A_1 \wedge A_2}\ (\wedge) \qquad \frac{\Sigma \ \vdash\ t : A_i}{\Sigma \ \vdash\ \sigma_i(t) : A_1 \vee A_2}\ (\vee)$$

$$\frac{\Sigma,\ x : A \ \vdash\ c : \bot}{\Sigma \ \vdash\ \lambda x.c \ :\ A^\bot}\ (\bot) \qquad \frac{\Sigma \ \vdash\ p : A \qquad \Sigma \ \vdash\ q : A^\bot}{\Sigma \ \vdash\ (p \ * \ q) : \bot}\ (\text{cut})$$

FIGURE 20. Typing rules of the system $\mathcal{B}$

The symmetry in classical calculi blocks a straightforward adaptation of the traditional reducibility technique which uses the fact that function types are "higher" in a natural sense than argument types, permitting semantic definitions to proceed by induction on types. In this paper we adapt the symmetric candidates technique to the intersection-types setting. As we can see, this technique applies generally to all of the symmetric proof-calculi we have investigated, including the $\overline{\lambda}\mu\widetilde{\mu}$-calculus of Curien and Herbelin [34, 51], and the Dual calculus of Wadler [147, 148].

The key to the symmetric candidates technique is to interpret types in certain families of *saturated* sets which are closed under inverse β-reduction. The problem in the intersection types setting arises since in standard semantics of intersection types, the interpretation of an intersection type $(A \cap B)$ is the intersection of the interpretations of A and B and in general *intersections of saturated sets are not saturated,*

A consequence of this fact is that the standard typing rule for intersection-introduction is not sound. So our type system has an intersection-elimination rule only. This is not a problem since intersection-introduction is not needed for characterising termination. In the absence of intersection-introduction, terms receive a type which is an intersection by double-negation elimination.

Theorem (Main result)**.** *A λ^{sym} term is terminating if and only if it is typable in $\mathcal{B}$.*

The direction "every terminating term is typable" follows the standard pattern from traditional λ-calculus where the standard intersection-introduction typing rule is not needed.

The proof that every typable term is terminating is analogous to the one given for [51] and [52]. We only briefly account for the differences.

We consider pairs $\{X_0, X_1\}$ which are *stable* if for every $r \in X_0$ and every $e \in X_1$, the command $(r * e)$ is terminating. They are *saturated* if for each i,

whenever $\lambda x.c$ satisfies: $\forall e \in X_i, c[x := e]$ is terminating, then $\lambda x.c \in X_{1-i}$.

An expression is *simple* if it is not a 'l-abstraction; a set X is simple if each term in X is simple.

We define the map $\Phi : 2^\Lambda \to 2^\Lambda$ by

$$\Phi(X) = \{e \mid e \text{ is of the form } \lambda x.c \text{ and } \forall r \in X, c[x := r] \text{ is terminating}\}$$
$$\cup \{e \mid e \text{ is simple and } \forall r \in X, (r * e) \text{ is terminating}\}.$$

If $X \neq \emptyset$ then $\Phi(X)$ is a set of terminating terms and if $X \subseteq SN$ then all variables are in $\Phi(X)$. Φ is antimonotone, hence $(\Phi \circ \Phi) = \Phi^2$ is monotone and has a complete lattice of fixed points, ordered by set inclusion.

If X is a simple set of terminating terms we denote by $X^\uparrow$ the least fixed point of Φ^2 with the property that $X \subseteq X^\uparrow$. Furthermore, let Fix_{Φ^2} be the set of fixed points of the operator Φ^2. If $R_1, \ldots, R_k$ are fixed points of Φ^2, let $(R_1 \curlywedge \ldots \curlywedge R_k)$ denote the meet of these elements in the lattice Fix_{Φ^2}.

Interpretation of types For each type T we define the set $[\![T]\!]$ as follows.

(1) When T is $\bot$ then $[\![T]\!]$ is the set of terminating terms.
(2) When T is a type variable we set R to be the set of term variables, then construct the pair $(R^\uparrow, \Phi(R^\uparrow))$. We then take $[\![T]\!]$ to be $R^\uparrow$ and $[\![T^\bot]\!]$ to be $\Phi(R^\uparrow)$.
(3) Suppose T is $(A_1 \wedge A_2)$. Set R to be $\{\langle t_1, t_2\rangle \mid t_i \in [\![A_i]\!],\ i = 1, 2\}$. We then take $[\![T]\!]$ to be $(R^\uparrow)$ and $[\![T^\bot]\!] = [\![{A_1}^\bot \vee {A_2}^\bot]\!]$ to be $\Phi(R^\uparrow)$.
(4) When T is $(A_1 \cap A_2 \cdots \cap A_n)$, $n \geqslant 2$, we take $[\![T]\!]$ to be $([\![A_1]\!] \curlywedge \ldots \curlywedge [\![A_n]\!])$ and then take $[\![T^\bot]\!]$ to be $\Phi([\![T]\!])$.

Note that the interpretation $[\![A_1 \vee A_2]\!]$ of a disjunction-type is determined in part 3 above since any type $B_1 \vee B_2$ is the Dual of ${B_1}^\bot \wedge {B_2}^\bot$.

We prove the following, which is the key for proving the Type soundnes.

(1) $[\![T]\!]$ is a set of terminating terms.
(2) $[\![A_1 \wedge A_2]\!] \supseteq \{\langle t_1, t_2\rangle \mid t_i \in [\![A_i]\!],\ i = 1, 2\}$.
(3) $[\![A_1 \vee A_2]\!] \supseteq \{\sigma_1(p) \mid p \in [\![A_1]\!]\} \cup \{\sigma_2(p) \mid p \in [\![A_2]\!]\}$.
(4) $(\lambda x.c) \in [\![A]\!]$ if for all $e \in [\![A^\bot]\!]$ we have $c[x := e]$ terminates.
(5) $[\![(A_1 \cap \cdots \cap A_k)]\!] \subseteq ([\![A_1]\!] \cap \cdots \cap [\![A_k]\!])$.

Since each $[\![T]\!]$ consists of terminating expressions the following theorem implies that all typable expressions are terminating.

Theorem (Type soundness)**.** *If expression t is typable with type T then $t \in [\![T]\!]$.*

11. Application in programming language theory

11.1. Functional languages.

. λ-calculus The basic concept of programming languages is the concept of a function, more precisely of intensional (or computational) function considered as a composition of computational steps, i.e., as algorithms (or methods). A universal model of computational functions is Church's λ-calculus [27]. λ-calculus as a simple language is very convenient to describe the semantics of programming languages (it is even used as a core for the languages Lisp, Algol, Scheme, ML, Haskell, etc).

The λ-calculus exists in basically two main flavours: call-by-name (of which Haskell implements the call-by-need variant) and call-by-value (as in Scheme, ML, C, Java, etc). Call-by-name has been extensively studied (see e.g., Barendregt [10], Krivine [101]) and call-by-value reasonably well too.

. Classical λ-calculus The $\lambda\mu$-calculus is an extension of λ-calculus with an operator similar to the `call-cc` operator that can be found in Scheme and ML. It also models weaker operators, such as `break` and `return` in C and Java.

The $\lambda\mu$-calculus is the prototypical formulation of a classical λ-calculus. As λ-calculus, $\lambda\mu$-calculus exists in call-by-name and call-by-value variants, the latter being a rather intricate structure to study [60, 129].

The $\overline{\lambda}\mu\tilde{\mu}$-calculus [34] is an improvement over $\lambda\mu$-calculus. It is an elegant calculus that exhibits different forms of symmetries. One of them is a symmetry between call-by-name and call-by-value which allows to significantly reduce the syntactic complexity of the call-by-value calculus compared to $\lambda\mu$-calculus.

. Call-by-value and call-by-name delimited continuation Historically, delimited control came with ad hoc operators for composing continuations: Felleisen [59] had a calculus that included a control operator *control* a delimiter *prompt* (denoted by $\mathcal{F}$ and #, respectively); Danvy and Filinski [36] had an operator `shift` to compose continuations and an operator `reset` to delimit them (these were also written $\mathcal{S}$ and $< \_ >$). Control operators are connected to classical logic, as first investigated by Griffin [90].

From [64], it is known that `shift` and `reset` are equivalent to the combination of Scheme's `call-cc`, Felleisen's *abort* and `reset`, and hence equivalent to $\mathcal{C}$ and *reset*. From [25], it is known that *control* and *prompt* are also equivalent to *shift* and *reset*, in spite that *control* is semantically more complex to study than $\mathcal{C}$ or `shift`. The simplicity of the semantics of `shift` together with its relevance for some programming applications contributed to set `shift` as a reference in delimited control. And this is so in spite (it seems that) it has never been studied until now as part of a dedicated λ-calculus of delimited control.

As shown by Ariola et al. [4], a fine-grained $\lambda\mu\widehat{\mathtt{tp}}$-calculus of delimited control of the strength of `shift` and `reset` is obtained if one starts from $\lambda\mu$-calculus and extends it first by a notation `tp` for the "toplevel" continuation, then by a toplevel delimiter. A possible interpretation for this toplevel delimiter is as a *dynamic* binder of `tp`, what justifies to interpret the resulting call-by-value calculus, called as an extension of call-by-value $\lambda\mu$-calculus with a single dynamically bound continuation variable $\widehat{\mathtt{tp}}$, where the hat on `tp` emphasises the dynamic treatment of the variable. A typical analogy for the dynamic continuation variable here is exception handling: each call to $\widehat{\mathtt{tp}}$ is dynamically bound to the closest surrounding $\widehat{\mathtt{tp}}$ binder, in exactly the same way as a raised exception is dynamically bound to the closest

surrounding handler. The expressiveness of this calculus was shown by simulating the operational semantics of `shift` and `reset` and of most standard control operators, such as E and A (*abort*) of Felleisen's, `call/cc` (the implementation of `call-cc` in Scheme).

$$\begin{array}{lcl}
\mathcal{S}\ M & \triangleq & \mu\alpha.[\widehat{\mathsf{tp}}](M\ \lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x) \\
< M > & \triangleq & \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]M \\
\mathcal{A}\,M & \triangleq & \mu_.[\widehat{\mathsf{tp}}]M \\
\mathcal{C}\ (\lambda k.M) & \triangleq & \mu\alpha_k.[\widehat{\mathsf{tp}}](M\ \lambda x.\mu_.[\alpha_k]x) \\
\texttt{call/cc}\ (\lambda k.M) & \triangleq & \mu\alpha_k.[\alpha_k](M\ \lambda x.\mu_.[\alpha_k]x)
\end{array}$$

$< M >$ ili $\langle M \rangle$?

$$\begin{array}{rcl}
\mathcal{S}\ M & \triangleq & \mu\alpha.[\widehat{\mathsf{tp}}](M\ \lambda x.\mu\widehat{\mathsf{tp}}.[\alpha]x) \\
< M > & \triangleq & \mu\widehat{\mathsf{tp}}.[\widehat{\mathsf{tp}}]M \\
\mathcal{A}\,M & \triangleq & \mu_.[\widehat{\mathsf{tp}}]M \\
\mathcal{C}\ (\lambda k.M) & \triangleq & \mu\alpha_k.[\widehat{\mathsf{tp}}](M\ \lambda x.\mu_.[\alpha_k]x) \\
\texttt{call/cc}\ (\lambda k.M) & \triangleq & \mu\alpha_k.[\alpha_k](M\ \lambda x.\mu_.[\alpha_k]x)
\end{array}$$

Herbelin and Ghilezan [94] proposed an approach to call-by-name delimited control. They devised a call-by-name variant of $\lambda\mu\widehat{\mathsf{tp}}$, for Ariola et al's call-by-value calculus of delimited control [4].

Continuation-passing-style semantics is given by a CPS transformation of the $\lambda\mu\widehat{\mathsf{tp}}$ into λ-calculus.

$$\begin{array}{lcl}
x^* & \triangleq & x \\
(\lambda x.M)^* & \triangleq & \lambda(x,k).M^*\,k \\
(M\,N)^* & \triangleq & \lambda k.M^*\,(N^*,k) \\
(\mu\alpha.c)^* & \triangleq & \lambda k_\alpha.c^* \\
([\alpha]M)^* & \triangleq & M^*\,k_\alpha \\
(\mu\widehat{\mathsf{tp}}.c)^* & \triangleq & c^* \\
([\widehat{\mathsf{tp}}]M)^* & \triangleq & M^*
\end{array}$$

FIGURE 21. Call-by-name CPS translation of $\lambda\mu\widehat{\mathsf{tp}}$

We present the behaviour of call-by-name $\lambda\mu\widehat{\mathsf{tp}}$ on standard examples that uses delimited control. We consider the example of list traversal that is used to emphasise the differences between Felleisen's operator $\mathcal{F}$ and *shift*. We extend $\lambda\mu\widehat{\mathsf{tp}}$ with a fixpoint operator, list constructors and a list destructor:

$$\begin{array}{lcl}
M,N & ::= & \ldots \mid \nu_x.M \mid [] \mid M{::}N \\
& \mid & \texttt{if}\ M\ \texttt{is}\ x{::}y\ \texttt{then}\ M\ \texttt{else}\ M
\end{array}$$

and we extend call-by-name reduction with the rules

$$\begin{array}{lcl}
\nu_x.M & \to & M[x := \nu_x.M] \\
\texttt{if } [] \texttt{ is } x{::}y \texttt{ then } M_2 \texttt{ else } M_1 & \to & M_1 \\
\texttt{if } M{::}N \texttt{ is } x{::}y \texttt{ then } M_2 \texttt{ else } M_1 & \to & M_2[x := M][y := N] \\
\texttt{if } \mu\alpha.c \texttt{ is } x{::}y \texttt{ then } M_2 \texttt{ else } M_1 & \to & \\
\qquad \mu\alpha.c[\alpha := [\alpha]\,(\texttt{if } \Box \texttt{ is } x{::}y \texttt{ then } M_2 \texttt{ else } M_1)] & & .
\end{array}$$

In informal ML syntax, the example is the following

```
let traverse l = let rec visit l = match l with
| [] -> []
| a::l' -> visit (shift (fun k -> a :: k l'))
                          in reset (visit l) in traverse [1;2;3]
```

Translated into $\Lambda\mu$, it gives

$$v\ (n_1{::}n_2{::}n_3{::}[])$$

where v is $\nu_f.(\lambda l.\texttt{if } l \texttt{ is } a{::}l' \texttt{ then } f\ (\mu\alpha.a{::}[\alpha]l') \texttt{ else } [])$. Translated into $\lambda\mu\widehat{\texttt{tp}}$, v is

$$\nu_f.(\lambda l.\texttt{if } l \texttt{ is } a{::}l' \texttt{ then } f\ (\mu\alpha.[\widehat{\texttt{tp}}]a{::}\mu\widehat{\texttt{tp}}.[\alpha]l') \texttt{ else } []).$$

Let ϵ be an arbitrary continuation distinct from $\widehat{\texttt{tp}}$. We write l_i for $n_i{::}\ldots{::}n_3{::}[]$. We list the steps of the reduction of $[\epsilon](v\ l_1)$:

$$\begin{array}{l}
[\epsilon]v\ l_1 \\
\to\ [\epsilon](\lambda l.\texttt{if } l \texttt{ is } a{::}l' \texttt{ then } v\ (\mu\alpha.a{::}[\alpha]l') \texttt{ else } [])\ l_1 \\
\to\ [\epsilon]\texttt{if } l_1 \texttt{ is } a{::}l' \texttt{ then } v\ (\mu\alpha.a{::}[\alpha]l') \texttt{ else } []) \\
\to\ [\epsilon]v\ (\mu\alpha.n_1{::}[\alpha]l_2) \\
\twoheadrightarrow \texttt{if } (\mu\alpha.n_1{::}[\alpha]l_2) \texttt{ is } a{::}l' \texttt{ then } v\ (\mu\alpha.a{::}[\alpha]l') \texttt{ else } [] \\
\to\ [\epsilon]\mu\alpha.n_1{::}[\alpha](\texttt{if } l_2 \texttt{ is } a{::}l' \texttt{ then } v\ (\mu\alpha.a{::}[\alpha]l') \texttt{ else } []) \\
\to\ n_1{::}[\epsilon](\texttt{if } l_2 \texttt{ is } a{::}l' \texttt{ then } v\ (\mu\alpha.a{::}[\alpha]l') \texttt{ else } []) \\
\to\ n_1{::}[\epsilon](v\ (\mu\alpha.n_2{::}[\alpha]l_3)) \\
\twoheadrightarrow n_1{::}[\epsilon](\mu\alpha.n_2{::}[\alpha](v\ l_3)) \\
\twoheadrightarrow n_1{::}n_2{::}[\epsilon](v\ l_3) \\
\twoheadrightarrow n_1{::}n_2{::}[\epsilon](\mu\alpha.n_3{::}[\alpha](v\ [])) \\
\twoheadrightarrow n_1{::}n_2{::}n_3{::}[\epsilon](v\ []) \\
\twoheadrightarrow n_1{::}n_2{::}n_3{::}[\epsilon][]
\end{array}$$

Otherwise said, the list traversal program copies its argument and shifts its continuation to the tail of the list.

11.2. **Object-oriented languages.** The aim of the following works was to give the basis for designing a calculus that combines class-based features with object-based ones. We propose two extensions of the "Core Calculus of Classes and Mixins" of [22], one with higher-order, composable mixins, the second one with incomplete objects.

Mixins [24] are subclass definitions parameterised over a superclass and were introduced as an alternative to some forms of multiple inheritance. A mixin can

be seen as a function that, given one class as an argument, produces a subclass, by adding and/or overriding certain sets of methods.

The calculus proposed in Bettini et al. [13, 14] extends the core calculus of classes and mixins of [22] with *higher-order* mixins. In this extension a mixin can: (*i*) be applied to a class to create a fully-fledged subclass; (*ii*) be *composed* with another mixin to obtain yet another mixin with more functionalities. In what we believe is quite a general framework, we give directions for designing a programming language equipped with higher-order mixins, although our study is not based on any actual object-oriented language.

In the calculus proposed in Bettini et al. [18, 17, 16, 15] we extend the core calculus of classes and mixins of [22] with *incomplete objects*. In addition to standard class instantiation, it is also possible to *instantiate mixins* thus obtaining *incomplete objects*.

Incomplete objects can be completed in two ways: (*i*) via *method addition*,

(*ii*) via *object composition*, that composes an incomplete object with a complete one that contains all the required methods. When a method is added, it becomes an effective component of the host object, meaning that the methods of the host object may invoke it, but also the new added method can use any of its sibling methods. The type system ensures that all method additions and object compositions are type safe and that only "complete" methods are invoked on objects. This way the type information at the mixin level is fully exploited, obtaining a "tamed" and safe object-based calculus.

The metatheory of both extensions is studied in Likavec [106]. In particular, the soundness property is proved, to guarantee the absence of run-time "message-not-understood" errors.

In addition, in [18] the calculus is endowed with width subtyping on complete objects, which provides enhanced flexibility while avoiding possible conflicts between method names.

Part 3 – Related work

. Related work on computational interpretation of logic The $\lambda\mu$-calculus of Parigot [117] embodies a Curry–Howard correspondence for classical natural deduction. It was introduced in call-by-name style, followed by a call-by-value variant, proposed by Ong and Stewart [116].

Herbelin [92] proposed the first "sequent" λ-calculus, named $\bar{\lambda}$, for which bijective correspondence between normal simply typed terms and cut-free proofs of the appropriate restriction of the Gentzen's LJ was obtained. He considered a λ-calculus with an explicit operator of substitution and substitution propagation rules. Each cut-elimination step corresponds to β-reduction, a substitution propagation or concatenation. However, this bijection failed to extend to sequent calculus with cuts.

After that, intuicionistic sequent λ-calculi were proposed by several authors, Barendregt and Ghilezan [12], Dyckhoff and Pinto [55], Espirito Santo and Pinto [58], among others.

The $\overline{\lambda}\mu\widetilde{\mu}$-calculus of Curien and Herbelin [34] provides a symmetric computational interpretation of classical sequent style logic. Expressions in $\overline{\lambda}\mu\widetilde{\mu}$ represent derivations in the sequent calculus proof system and reduction reflects the process of cut-elimination. This calculus provides an environment for a more fine-grained analysis of calculations within languages with control operators. Since its introduction, Curien and Herbelin's calculus has had a strong influence on the further understanding between calculi with control operators and classical logic (see for example [3, 2, 147, 148]).

In the calculus of Urban and Bierman [143, 144] derivations correspond exactly to cut elimination.

This calculus inspired Lengrand's $\lambda\xi$ in [103] and further led to the development of $\mathcal{X}$ calculus of van Bakel et al. [5], and van Bakel and Lescanne [146]. In this work, a calculus which interprets directly the implicational sequent logic is proposed as a language in which many kinds of other calculi can be implemented, from λ-calculus to $\overline{\lambda}\mu\widetilde{\mu}$ through a calculus of explicit substitution and $\lambda\mu$.

Wadler's dual calculus [147, 148] corresponds to Gentzen's classical sequent calculus. Conjunction, disjunction, and negation are primitive, whereas implication is defined in terms of the other connectives.

One of the most recently proposed systems is λ^{Gtz}-calculus, developed by Espirito Santo [56], whose simply typed version corresponds to the sequent calculus for intuicionistic implicational logic.

Prior to Curien and Herbelin's $\overline{\lambda}\mu\widetilde{\mu}$ [34] several term-assignment systems for sequent calculus were proposed as a tool for studying the process of cut-elimination [122, 12, 143]. In these systems—with the exception of the one in [143]—expressions do not unambiguously encode sequent derivations.

The Symmetric Lambda Calculus of Barbanera and Berardi [6], although not based on sequent calculus, belongs in the tradition of exploiting the symmetries found in classical logic, in their case with the goal of extracting constructive content from classical proofs.

. Related work on strong normalisation Barbanera and Berardi [6] proved SN for their calculus using a "symmetric candidates" technique; Urban and Bierman [143] adapted their technique to prove SN for their sequent-based system. Lengrand [103] shows how simply-typed $\overline{\lambda}\mu\widetilde{\mu}$ and the calculus of Urban and Bierman [143] are mutually interpretable, so that the strong normalisation proof of the latter calculus yields another proof of strong normalisation for simply-typed $\overline{\lambda}\mu\widetilde{\mu}$. Polonovski [121] presents a proof of SN for $\overline{\lambda}\mu\widetilde{\mu}$ with explicit substitutions using the symmetric candidates idea of Barbanera and Berardi [6]. Pym and Ritter [125] identify two forms of disjunction for Parigot's $\lambda\mu$-calculus [117]; they prove strong normalisation for $\lambda\mu\nu$-calculus ($\lambda\mu$-calculus extended with such disjunction). David and Nour [37] give an arithmetical proof of strong normalisation for a symmetric $\lambda\mu$-calculus.

The larger context of related research includes a wealth of work in logic and programming languages. In the 1980's and early 1990's Reynolds explored the role that intersection types can play in a practical programming language (see for example the report [127] on the language Forsythe).

. Related work on continuation semantics Continuation-passing-style (cps) translations were introduced by Fischer and Reynolds in [65] and [126] for the call-by-value λ-calculus, whereas a call-by-name variant was introduced by Plotkin in [120]. Moggi gave a semantic version of a call-by-value cps translation in his study of notions of computation in [113]. Lafont [102] introduced a cps translation of the call-by-name $\lambda\mathcal{C}$-calculus [61, 62] to a fragment of λ-calculus that corresponds to the $\neg, \wedge$-fragment of the intuitionistic logic. Hence, continuation semantics can be seen as a generalization of the double negation rule from logic, in a sense that cps translation is a transformation on terms which, when observed on types, corresponds to a double negation translation.

As early as 1989 Filinsky [63] explored the notion that the reduction strategies call-by-value and call-by-name could be dual to each other in the presence of continuations. Filinski defined a symmetric λ-calculus in which values and continuations comprised distinct syntactic sorts and whose denotational semantics expressed the call-by-name vs call-by-value duality in a precise categorical sense.

Categorical semantics for both, call-by-name and call-by-value versions of Parigot's $\lambda\mu$-calculus [117] with disjunction types was given by Selinger in [133]. In this work the notion of *control category* is formally introduced and formalised as an extension of cartesian closed category with premonoidal structure. It is showed that the call-by-name $\lambda\mu$-calculus forms an internal language for control categories, whereas the call-by-value $\lambda\mu$-calculus forms an internal language for co-control categories. The opposite of the call-by-name model is shown to be equivalent to the call-by-value model in the presence of product and disjunction types. Hofmann and Streicher presented categorical continuation models for the call-by-name $\lambda\mu$-calculus in [96] and showed the completeness.

Lengrand gave categorical semantics of the *typed* $\overline{\lambda}\mu\widetilde{\mu}$-calculus and the $\lambda\xi$-calculus (implicational fragment of the classical sequent calculus LK) in [103].

Ong [115] defined a class of categorical models for the call-by-name $\lambda\mu$-calculus based on fibrations. This model was later extended for two forms of disjunction by Pym and Ritter in [125].

References

[1] R. M. Amadio and P.-L. Curien, *Domains and lambda-calculi*, Cambridge University Press, Cambridge, 1998.

[2] Z. M. Ariola, H. Herbelin, and A. Sabry, *A type-theoretic foundation of continuations and prompts*; in: *Proc. 9th Internat. Conf. on Functional Programming ICFP '04*, pp 40–53, 2004.

[3] Z. M. Ariola and H. Herbelin, *Minimal classical logic and control operators*; in: *Proc. Annual International Colloquium on Automata, Languages and Programming ICALP '03*, Lect. Notes Comput. Sci. 2719, Springer-Verlag, 2003, pp. 871–885.

[4] Z. M. Ariola, H. Herbelin, and A. Sabry, *A type-theoretic foundation of delimited continuations*, High.-Order Symb. Comput. 2007, to appear.

[5] S. v. Bakel, S. Lengrand, and P. Lescanne, *The language $\mathcal{X}$: circuits, computations and classical logic*; in: *Proc. 9th Italian Conf. on Theoretical Computer Science ICTCS '05*, Lect. Notes Comput. Sci. 3701, Springer-Verlag, 2005, pp. 81–96.

[6] F. Barbanera and S. Berardi, *A symmetric lambda calculus for classical program extraction*, Inf. Comput. 125(2):103–117, 1996.

[7] F. Barbanera, M. Dezani-Ciancaglini, and U. de' Liguoro, *Intersection and union types: syntax and semantics*, Inf. Comput. 119(2):202–230, 1995.

[8] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro, *Intersection and union types: Syntax and semantics*, Inf. Comput. 119(2):202–230, 1995.

[9] H. P. Barendregt, *Lambda calculi with types*; in: S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pp. 117–309 Oxford University Press, Oxford, 1992.

[10] H. P. Barendregt, *The Lambda Calculus: its Syntax and Semantics*, revised edition, North-Holland, Amsterdam, 1984.

[11] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini, *A filter lambda model and the completeness of type assignment*, J. Symb. Log. 48(4):931–940 (1984), 1983.

[12] H. P. Barendregt and S. Ghilezan, *Lambda terms for natural deduction, sequent calculus and cut-elimination*, J. Funct. Program. 10(1):121–134, 2000.

[13] L. Bettini, V. Bono, and S. Likavec, *A core calculus of higher-order mixins and classes*; in: *Proc. Workshop Types for Proofs and Programs TYPES '03 (Selected Papers)*, Lect. Notes Comput. Sci. 3085, pp. 83–98, Springer-Verlag, 2004.

[14] L. Bettini, V. Bono, and S. Likavec, *A Core Calculus of Higher-Order Mixins and Classes*; in: *Proc. 19th Annual ACM Symposium on Applied Computing, SAC '04*, pages 1508–1509 ACM Press, 2004.

[15] L. Bettini, V. Bono, and S. Likavec, *A core calculus of mixin-based incomplete objects*; in: *Proc. 11th International Workshop on Foundations of Object-Oriented Languages, FOOL '04*, pp. 29–41, 2004.

[16] L. Bettini, V. Bono, and S. Likavec, *A core calculus of mixins and incomplete objects*; in: *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications OOPSLA '04*, pp. 208–209, ACM Press, 2004.

[17] L. Bettini, V. Bono, and S. Likavec, *Safe and Flexible Objects*; in: *Proc. 20th Annual ACM Symposium on Applied Computing SAC '05*, pp. 1258–1263, ACM Press, 2005.

[18] L. Bettini, V. Bono, and S. Likavec, *Safe and Flexible Objects with Subtyping*, J. Object Technology 4(10), 2005, Special Issue on "The 20th ACM SAC - March 2005".

[19] G. M. Bierman, *A computational interpretation of the $\lambda\mu$-calculus*; in: *Proc. Symp. on Mathematical Foundations of Computer Science MFCS '98*, Lect. Notes Comput. Sci. 1450, pp. 336–345 Springer-Verlag, 1998.

[20] C. Böhm, *Alcune proprieta delle forme $\beta-\eta$-normali nel $\lambda-k$-calcolo*, Publ. Inst. Appl. Calc. 696:1–19, 1968.
[21] C. Böhm and M. Dezani-Ciancaglini, *λ-terms as total or partial functions on normal forms*, in: *λ-calculus and Computer Science Theory*, Lect. Notes Comput. Sci. 37, pp. 96–121, Springer-Verlag 1975.
[22] V. Bono, A. Patel, and V. Shmatikov, *A core calculus of classes and mixins*, in: *Proc European Conf. on Object-Oriented Programming ECOOP '99*, Lect. Notes Comput. Sci. 1628, pp. 43–66. Springer-Verlag, 1999.
[23] V. Bono, B. Venneri, and L. Bettini, *A typed lambda calculus with intersection types*, Theor. Comput. Sci. 398(1-3):95–113, 2008.
[24] G. Bracha and W. Cook, *Mixin-based inheritance*, in: *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications OOPSLA '90*, pp. 303–311, 1990.
[25] C.-c. Shan, *Shift to control*, in: *Proc. 5th Workshop on Scheme and Functional Programming*, pp. 99–107, 2004.
[26] S. Carlier and J. B. Wells, *Type inference with expansion variables and intersection types in system E and an exact correspondence with beta-reduction*, in: *Proc. 6th Conf. on Principles and Practice of Declarative Programming PPDP '04*, pp. 132–143, ACM, 2004.
[27] A. Church, *A set of postulates for the foundation of logic*, Ann. Math. II.33:346–366, 1932.
[28] A. Church, *A formulation of the simple theory of types*, J. Symb. Log. 5:56–68, 1940.
[29] M. Coppo and M. Dezani-Ciancaglini, *A new type-assignment for lambda terms*, Archiv Math. Logik 19:139–156, 1978.
[30] M. Coppo and M. Dezani-Ciancaglini, *An extension of the basic functionality theory for the λ-calculus*, Notre Dame J. Formal Logic 21(4):685–693, 1980.
[31] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri, *Principal type schemes and λ-calculus semantics*; in: J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 535–560, Academic Press, London, 1980.
[32] P.-L. Curien, *Abstract machines, control, and sequents*, in: *Applied Semantics, International Summer School, APPSEM 2000, Advanced Lectures*, Lect. Notes Comput. Sci. 2395, pp. 123–136, Springer-Verlag, 2002.
[33] P.-L. Curien, *Symmetry and interactivity in programming*, Bull. Symb. Log. 9(2):169–180, 2003.
[34] P.-L. Curien and H. Herbelin, *The duality of computation*; in: *Proc. 5th Internat. Conf. on Functional Programming, ICFP'00*, pp. 233–243, Montreal, Canada, 2000; ACM Press.
[35] H. B. Curry, J. R. Hindley, and J. P. Seldin, *Combinatory Logic*, volume II, North-Holland, Amsterdam, 1972.
[36] O. Danvy and A. Filinski, *A functional abstraction of typed contexts*, Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, Aug. 1989.
[37] R. David and K. Nour, *Arithmetical proofs of strong normalization results for the symmetric $\lambda\mu$-calculus*; in: *Proc. Typed Lambda Calculus and Application, TLCA '05*, Lect. Notes Comput. Sci. 3461, pp. 162–178, Springer-Verlag, 2005.
[38] R. David and W. Py, *Lambda-mu-calculus and Böhm's theorem*, J. Symb. Log. 66(1):407–413, 2001.
[39] P. de Groote, *A CPS-translation of the $\lambda\mu$-calculus*, in: *Proc. of the Colloquium on Trees in Algebra and Programming, CAAP '94*, Lect. Notes Comput. Sci. 787, pp. 85–99, Springer-Verlag, 1994.
[40] P. de Groote, *On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control*; in: *Proc. Internat. Conf. on Logic Programming and Automated Reasoning, LPAR '94*, Lect. Notes Comput. Sci. 822, pp. 31–43, Springer-Verlag, 1994.
[41] M. Dezani-Ciancaglini and S. Ghilezan, *A lambda model characterizing computational behaviours of terms*, in: *Proc. International Workshop on Rewriting in Proof and Computation RPC '01*, pp. 100–119, 2001.

[42] M. Dezani-Ciancaglini and S. Ghilezan, *A behavioural lambda model*, Schedae Informaticae Universitas Iagelonica, 12:35–47, 2003.

[43] M. Dezani-Ciancaglini and S. Ghilezan, *Two behavioural lambda models*; in: *Proc. Workshop Types for Proofs and Programs TYPES '02*, Lect. Notes Comput. Sci. 2646, pp. 127–147, Springer-Verlag, 2003.

[44] M. Dezani-Ciancaglini, S. Ghilezan, and S. Likavec, *Behavioural inverse limit models*, Theor. Comput. Sci. 316(1–3):49–74, 2004.

[45] M. Dezani-Ciancaglini, S. Ghilezan, and B. Venneri, *The "relevance" of intersection and union types*, Notre Dame J. Formal Logic, 38(2):246–269, 1997.

[46] M. Dezani-Ciancaglini and E. Giovannetti, *From Böhm teorem to observational equivalence: an informal account*; in: *Proc. Böhm theorem: applications to Computer Science Theory – BOTH '01*, Electr. Notes Theor. Comput. Sci. 50(2), 2001.

[47] M. Dezani-Ciancaglini, F. Honsell, and Y. Motohama, *Compositional characterization of λ-terms using intersection types*; in: *Proc. Mathematical Foundations of Computer Science MFCS '00*, Lect. Notes Comput. Sci. 1893, pp. 304–314, Springer-Verlag, 2000.

[48] D. Dougherty, S. Ghilezan, and P. Lescanne, *Characterizing strong normalization in a language with control operators*; in: *Proc. 6th Conf. on Principles and Practice of Declarative Programming PPDP '04*, pp. 155–166, ACM Press, 2004.

[49] D. Dougherty, S. Ghilezan, and P. Lescanne, *Intersection and union types in the $\overline{\lambda}\mu\widetilde{\mu}$-calculus*; in: *Proc. Workshop on Intersection Types and Related Systems ITRS '04*, Electr. Notes Theor. Comput. Sci. 136:153–172, 2005.

[50] D. Dougherty, S. Ghilezan, and P. Lescanne, *A general technique for analyzing termination in symmetric proof calculi*; in: *Proc. 9th International Workshop on Termination WST '07*, 2007.

[51] D. Dougherty, S. Ghilezan, and P. Lescanne, *Characterizing strong normalization in the Curien-Herbelin symmetric lambda calculus: extending the Coppo-Dezani heritage*; Theor. Comput. Sci. 398:114–128, 2008.

[52] D. Dougherty, S. Ghilezan, P. Lescanne, and S. Likavec, *Strong normalization of the dual classical sequent calculus*; in: *Proc. 12th Internat. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning LPAR '05*, Lect. Notes Comput. Sci. 3835, pp. 169–183 Springer-Verlag, 2005.

[53] K. Došen and Z. Petrić, *The typed Böhm teorem*; in: *Proc. Böhm theorem: applications to Computer Science Theory – BOTH '01*, Electr. Notes Theor. Comput. Sci. 50(2), p.13, 2001.

[54] J. Dunfield and F. Pfenning, *Type assignment for intersections and unions in call-by-value languages*; in: *Proc. 6th Internat. Conf. on Foundations of Software Science and Computation Structures FOSSACS '03*, Lect. Notes Comput. Sci. 2620, pp. 250–266, Springer-Verlag, 2003.

[55] R. Dyckhoff and L. Pinto, *Cut-elimination and a permutation-free sequent calculus for intuitionistic logic*, Studia Logica 60(1):107–118, 1998.

[56] J. Espírito Santo, *Completing Herbelin's programme*; in: *Proc. Conf. on Typed Lambda Calculus and Applications TLCA '07*, Lect. Notes Comput. Sci. 4583, pp. 118–132, Springer-Verlag, 2007.

[57] J. Espírito Santo, S. Ghilezan, and J. Ivetić, *Characterising strongly normalising intuitionistic sequent terms*, in: *Proc. Workshop Types for Proofs and Programs TYPES '07 (Selected Papers)*, Lect. Notes Comput. Sci. 4941, pp. 85-99 Springer-Verlag, 2007.

[58] J. Espírito Santo and L. Pinto, *Permutative conversions in intuitionistic multiary sequent calculi with cuts*; in: *Proc. Conf. on Typed Lambda Calculus and Applications TLCA '03*, Lect. Notes Comput. Sci.2071, pp. 286–300, , Springer-Verlag, 2003.

[59] M. Felleisen, *The theory and practice of first-class prompts*, in: *Proc. 15th ACM Symp. on Principles of Programming Languages POPL '88*, pp. 180–190. ACM, 1988.

[60] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. F. Duba, *Reasoning with continuations*, in: *Proc. 1st Symposium on Logic in Computer Science LICS '86*, pp. 131–141, 1986.

[61] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. F. Duba, *A syntactic theory of sequential control*, Theor. Comput. Sci. 52(3):205–237, 1987.
[62] M. Felleisen and R. Hieb, *The revised report on the syntactic theories of sequential control and state*, Theor. Comput. Sci., 103(2):235–271, 1992,
[63] A. Filinski, *Declarative continuations and categorical duality*, Master's thesis, DIKU, Computer Science Department, University of Copenhagen, 1989, DIKU Rapport 89/11.
[64] A. Filinski, *Representing monads*, in: *Proc. 21st ACM Symp. on Principles of Programming Languages, POPL'94*, pp. 446–457 ACM, 1994.
[65] M. Fischer, *Lambda calculus schemata*, in: *Proc. ACM Conf. on Proving Assertions Ab out Programs '72*, pp. 104–109 ACM Press, 1972.
[66] G. Frege, *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought*, Halle, 1879; reprinted in Jan van Heijenoort, editor, *From Frege to Gödel, A Sourcebook in Mathematical Logic*, 1879–1931, Harvard University Press, 1967.
[67] J. H. Gallier, *Typing untyped λ-terms, or reducibility strikes again!*, Ann. Pure Appl. Logic 91:231–270, 1998.
[68] J. H. Gallier, *Constructive logics part i: A tutorial on proof systems and typed lambda-calculi*, Theor. Comput. Sci. 110(2):249–339, 1993.
[69] G. Gentzen, *Unterschungen über das logische Schliessen*, Math Z. 39 (1935), 176–210; in: M. Szabo, editor, *Collected papers of Gerhard Gentzen*, pp. 68–131, North-Holland, 1969.
[70] S. Ghilezan, *Inhabitation in intersection and union type assignment systems*, J. Log. Comput. 3(6):671–685, 1993.
[71] S. Ghilezan, *Application of typed lambda calculi in the untyped lambda calculus*, in: *Proc. Logical Foundations of Computer Science LFCS '04*, Lect. Notes Comput. Sci. 813, pp. 129–139, 1994.
[72] S. Ghilezan, *Generalized finitness of developments in typed lambda calculi*, J. Autom. Lang. Comb., 1(4):247–258, 1996.
[73] S. Ghilezan, *Strong normalization and typability with intersection types*, Notre Dame J. Formal Logic 37(1):44–52, 1996.
[74] S. Ghilezan, *Cut elimination in the simply typed lambda calculus*, in: *Proc. 1st Panhellenic Logic Symposium PLS '97*, pp. 21–24, 1997.
[75] S. Ghilezan, *Natural deduction and sequent typed lambda calculus*, Novi Sad J. Math. 29:209–220, 1999.
[76] S. Ghilezan, *Topologies in lambda calculus*, in: *Proc. 2nd Panhellenic Logic Symposium PLS '99*, pp. 102–106, 1999.
[77] S. Ghilezan, *Intersection types and topologies and lambda calculus*, in: *ICALP Satellite Workshops*, pp. 303–304, 2000.
[78] S. Ghilezan, *Full intersection types and topologies in lambda calculus*, J. Comput. Sys. Sci. 62(1):1–14, 2001.
[79] S. Ghilezan, *Types and confluence in lambda calculus*, in: *Proc. 3rd Panhellenic Logic Symposium PLS '01*, 2001.
[80] S. Ghilezan, *Terms for natural deduction, sequent calculus and cut elimination in classical logic*; in: *Reflections on Type Theory, Lambda Calculus, and the Mind – Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, 2007.
[81] S. Ghilezan and J. Ivetić, *Intersection types for λ^{gtz} calculus*, Publ. Inst. Math., Nouv. Sér. 82(96):85–91, 2007.
[82] S. Ghilezan and V. Kuncak, *Confluence of untyped lambda calculus via simple types*, in: *Proc. Italian Conf. on Theoretical Computer Science ICTCS '01*, Lect. Notes Comput. Sci. 2202, pp. 38–49, , Springer-Verlag, 2001.
[83] S. Ghilezan and V. Kunčak, *Reducibility method in simply typed lambda calculus*, Novi Sad J. Math. 31:27–32, 2001.
[84] S. Ghilezan, V. Kunčak, and S. Likavec, *Reducibility method for termination properties of typed lambda terms*, in: *Proc. 5th International Workshop on Termination WST '01*, pp. 14–16, 2001.

[85] S. Ghilezan and P. Lescanne, *Classical proofs, typed processes and intersection types*, in: *Proc. Workshop Types for Proofs and Programs TYPES '03 (Selected Papers)*, Lect. Notes Comput. Sci. 3085, pp. 226–241, Springer-Verlag, 2004.

[86] S. Ghilezan and S. Likavec, *Reducibility: A Ubiquitous Method in Lambda Calculus with Intersection Types*, in: *Proc. Workshop on Intersection Types and Related Systems ITRS '02*, Electr. Notes Theor. Comput. Sci. 70, 2003.

[87] S. Ghilezan and S. Likavec, *Extensions of the reducibility method*, in: *Proc. 4th Panhellenic Logic Symposium PLS 04*, pp. 107–112, 2004.

[88] J.-Y. Girard, *Une extension de l'interprétation de Gödel à l'analyse, et son application à l'elimination des coupures dans l'analyse et la théorie des types*, in: *Proc. 2nd Scandinavian Logic Symposium*, pp. 63–92. North-Holland, Amsterdam, 1971.

[89] J.-Y. Girard, *A new constrcutive logic: classical logic*, Math. Struct. Comput. Sci. 1(3):255–296, 1991.

[90] T. Griffin, *A formulae-as-types notion of control*, in: *Proc. 19th Annual ACM Symp. on Principles Of Programming Languages, POPL '90*, pp. 47–58, ACM Press, 1990.

[91] H. Herbelin, Private communication.

[92] H. Herbelin, *A lambda calculus structure isomorphic to Gentzen-style sequent calculus structure*, in: *Proc. Conf. on Computer Science Logic, CSL '94*, Lect. Notes Comput. Sci. 933, pp. 61–75, Springer-Verlag, 1995.

[93] H. Herbelin, *Séquents qu'on calcule : de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes*, Thèse, Université Paris 7 1995.

[94] H. Herbelin and S. Ghilezan, *An approach to call-by-name delimited continuations*, in: *Proc. 35th Annual ACM Symp. on Principles of Programming Languages POPL '08*, pp. 383–394, SIGPLAN Notices 43, ACM Press, 2008.

[95] J. R. Hindley, *Coppo–Dezani types do not correspond to propositional logic*, Theor. Comput. Sci. 28(1-2):235–236, 1984.

[96] M. Hofmann and T. Streicher, *Completeness of continuation models for λμ-calculus*, Inf. Comput. 179(2):332–355, 2002.

[97] W. A. Howard, *The formulas-as-types notion of construction*, in: J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490, Academic Press, 1980.

[98] F. Joachimski and R. Matthes, *Standardization and confluence for ΛJ*, in: *Proc. Rewriting Techniques and Applications RTA '00*, Lect. Notes Comput. Sci. 1833, pp. 141–155, Springer-Verlag, 2000.

[99] A. J. Kfoury and J. B. Wells, *Principality and type inference for intersection types using expansion variables*, Theor. Comput. Sci. 311(1-3):1–70, 2004.

[100] G. Koletsos, *Church-Rosser theorem for typed functionals*, J. Symb. Log. 50:782–790, 1985.

[101] J.-L. Krivine, *Lambda-calcul types et modèles*, Masson, Paris, 1990.

[102] Y. Lafont, *Negation versus implication*, Draft, 1991.

[103] S. Lengrand, *Call-by-value, call-by-name, and strong normalization for the classical sequent calculus*, Electr. Notes Theor. Comput. Sci. 86, Elsevier, 2003.

[104] S. Lengrand, P. Lescanne, D. Dougherty, M. Dezani-Ciancaglini, and S. van Bakel, *Intersection types for explicit substitutions*, Inf. Comput. 189(1):17–42, 2004.

[105] S. Likavec, *Reducibility method for λ calculus with intersection types*, Master's thesis, University of Novi Sad, Serbia, 2005.

[106] S. Likavec, *Types for object oriented and functional programming languages*, PhD thesis, Università di Torino, Italy and ENS Lyon, France, 2005.

[107] S. Likavec and P. Lescanne, *On untyped Curien-Herbelin calculus*, in: *Proc. 1st Workshop on Classical Logic and Computation CLaC '06*, 2006.

[108] L. Liquori and S. Ronchi Della Rocca, *Intersection-types à la church*, Inf. Comput. 205(9):1371–1386, 2007.

[109] R. Matthes, *Characterizing strongly normalizing terms of a calculus with generalized applications via intersection types*, in: *ICALP Satellite Workshops*, pp. 339–354, 2000.

[110] G. Mints, *Normal forms for sequent derivations*, in: P. Odifreddi, editor, *Kreiseliana. About and Around Georg Kreisel*, pp. 469–492. A. K. Peters, Wellesley, 1996.
[111] J. C. Mitchell, *Type systems for programming languages*, in: J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Volume B, pp. 415–431, Elsevier, Amsterdam, 1990.
[112] J. C. Mitchell, *Foundation for Programmimg Languages*, MIT Press, Boston, 1996.
[113] E. Moggi, *Notions of computations and monads*, Inf. Comput. 93(1), 1991.
[114] C. R. Murthy, *Classical proofs as programs: How, what, and why*, in: *Constructivity in Computer Science*, Lect. Notes Comput. Sci. 613, pp. 71–88, Springer-Verlag, 1991.
[115] C.-H. L. Ong, *A semantic view of classical proofs: type-theoretic, categorical, denotational characterizations*, in: *Proc. 11th IEEE Annual Symposium on Logic in Computer Science LICS '97*, pp. 230–241. IEEE Computer Society Press, 1997.
[116] C.-H. L. Ong and C. A. Stewart, *A Curry–Howard foundation for functional computation with control*, in: *Proc. 24th ACM Symp. on Principles of Programming Languages POPL '97*, pp. 215–227, 1997.
[117] M. Parigot, *An algorithmic interpretation of classical natural deduction*, in: *Proc. Internat. Conf. on Logic Programming and Automated Reasoning, LPAR '92*, Lect. Notes Comput. Sci. 624, pp. 190–201, Springer-Verlag, 1992.
[118] M. Parigot, *Proofs of strong normalisation for second order classical natural deduction*, J. Symb. Log. 62(4):1461–1479, 1997.
[119] B. C. Pierce, *Programming with intersection types, union types, and polymorphism*, Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb, 1991.
[120] G. D. Plotkin, *Call-by-name, call-by-value and the λ-calculus*, Theor. Comput. Sci. 1:125–159, 1975.
[121] E. Polonovski, *Strong normalization of $\lambda\mu\tilde{\mu}$-calculus with explicit substitutions*, in: *Proc. 7th Internat. Conf. on Foundations of Software Science and Computation Structures, FOSSACS '04*, Lect. Notes Comput. Sci. 2987, pp. 423–437, Springer-Verlag, 2004.
[122] G. Pottinger, *Normalization as homomorphic image of cut-elimination*, Ann. Math. Log. 12:323–357, 1977.
[123] G. Pottinger, *A type assignment for the strongly normalizable λ-terms*, in: J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 561–577, Academic Press, London, 1980.
[124] D. Prawitz, *Natural Deduction*, Almqvist and Wiksell, 1965.
[125] D. Pym and E. Ritter, *On the semantics of classical disjunction*, J. Pure Appl. Algebra 159:315–338, 2001.
[126] J. C. Reynolds, *Definitional interpreters for higher-order programming languages*, in: *Proc. ACM Annual Conference*, pp. 717–740, ACM Press, 1972.
[127] J. C. Reynolds, *Design of the programming language Forsythe*, Report CMU-CS-96-146, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.
[128] K. Rose, *Explicit substitutions: Tutorial and survey*, Technical Report LS-96-3, BRICS, 1996.
[129] A. Sabry and M. Felleisen, *Reasoning about programs in continuation-passing style*, Lisp and Symbolic Computation 6(3-4):289–360, 1993.
[130] P. Sallé, *Une extension de la théorie des types en lambda-calcul*, in: *Proc. 5th Internat. Conf. on Automata, Languages and Programming ICALP '79*, Lect. Notes Comput. Sci. 62, pp. 398–410, Springer-Verlag, 1978.
[131] A. Saurin, *Separation with streams in the λμ-calculus*, in: *Proc. 20th Annual IEEE Symp. on Logic in Computer Science LICS '05*, pp. 356–365 IEEE Computer Society Press, 2005.
[132] D. S. Scott, *Continuous lattices*, in: *Toposes, Algebraic Geometry and Logic*, Lect. Notes Math. 274, pp. 97–136, Springer-Verlag, 1972.
[133] P. Selinger, *Control categories and duality: On the categorical semantics of the lambda-mu calculus*, Math. Struct. Comput. Sci. 11(2):207–260, 2001.

[134] A. K. Simpson, *Categorical completeness results for simply typed lambda calculus*, in: *Proc. Conf. on Typed Lambda Calculus and Applications TLCA '95*, Lect. Notes Comput. Sci. 902, pp. 414–427, Springer-Verlag, 1995.
[135] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry–Howard isomorphism*, Studies in Logic and the Foundations of Mathematics, 149. Elsevier, 2006.
[136] R. Statman, *Completeness, invariance and λ-definability*, J. Symb. Log. 47(1):17–26, 1982.
[137] R. Statman, *Logical relations and the typed λ-calculus*, Inf. Control 65:85–97, 1985.
[138] T. Streicher and B. Reus, *Classical logic, continutation semantics and abstract machines*, J. Funct. Program. 8(6):543–572, 1998.
[139] G. J. Sussman and G. L. S. Jr, *Scheme: A interpreter for extended lambda calculus*, High.-Order Symb. Comput. 11(4):405–439, 1998.
[140] W. W. Tait, *Intensional interpretations of functionals of finite type I*, J. Symb. Log. 32:198–212, 1967.
[141] W. W. Tait, *A realizability interpretation of the theory of species*, in: *Logic Colloquium*, Lect. Notes Math. 453, pp. 240–251, Springer-Verlag, 1975.
[142] M. Takahashi, *Parallel reduction in λ-calculus*, Inf. Comput. 118:120–127, 1995.
[143] C. Urban and G. M. Bierman, *Strong normalisation of cut-elimination in classical logic*, in: *Proc. Conf. on Typed Lambda Calculus and Applications, TLCA '99*, Lect. Notes Comput. Sci. 1581, pp. 365–380 Springer-Verlag, 1999.
[144] C. Urban and G. M. Bierman, *Strong normalisation of cut-elimination in classical logic*, Fund. Inf. 45(1-2):123–155, 2001.
[145] S. van Bakel, *Intersection type assignment systems*, Theor. Comput. Sci. 38(2):246–269, 1997.
[146] S. van Bakel and P. Lescanne, *Computation with classical sequents*, Math. Struct. Comput. Sci. 18(3):555–609, 2008.
[147] P. Wadler, *Call-by-value is dual to call-by-name*, in: *Proc. 8th Internat. Conf. on Functional Programming ICFP '03*, pp. 189–201, 2003.
[148] P. Wadler, *Call-by-value is dual to call-by-name, reloaded*; in: *Proc. Conf. on Rewriting Technics and Applications RTA '05*, Lect. Notes Comput. Sci. 3467, pp. 185–203, 2005.